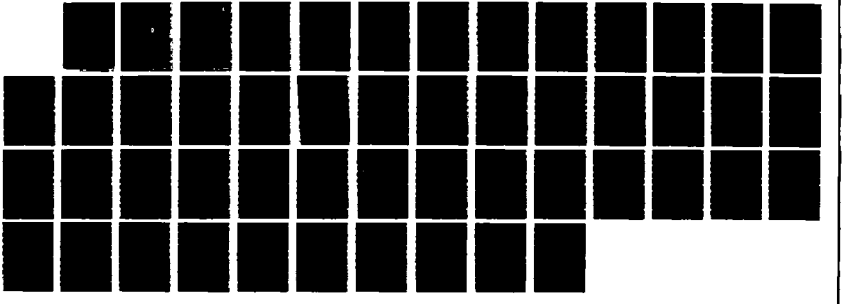
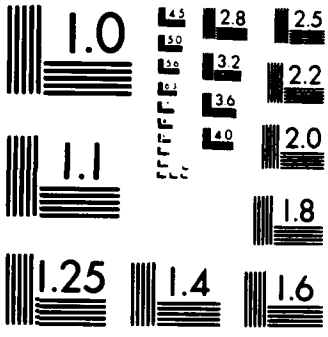
AD-A181 716 AN ANALYTICAL CACHE MODEL(U) STANFORD UNIV CA COMPUTER SYSTEMS LAB A AGARWAL ET AL SEP 86 CSL-TR-86-304 MDA903-83-C-0335 1/1

UNCLASSIFIED F/G 12/6 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-2192

AN ANALYTICAL CACHE MODEL

Anant Agarwal
Mark Horowitz
John Hennessy

Technical Report: CSL-TR-86-304

SEPTEMBER 1986

This work has been supported primarily by the Defense Advanced Research Projects Agency under contract number MDA903-83-C-0335.

COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-2192

AN ANALYTICAL CACHE MODEL

Anant Agarwal
Mark Horowitz
John Hennessy

Technical Report: CSL-TR-86-304

SEPTEMBER 1986

This work has been supported primarily by the Defense Advanced Research Projects Agency under contract number MDA903-83-C-0335.

AN ANALYTICAL CACHE MODEL

Anant Agarwal, Mark Horowitz and John Hennessy

Technical Report: CSL T.R. 86-304

September 1986

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

Trace driven simulation and hardware measurement are the techniques most often used to obtain accurate performance figures for caches. The former requires a large amount of simulation time to evaluate each cache configuration while the latter is restricted to measurements of existing caches. An analytical cache model that uses parameters extracted from address traces of programs can provide estimates of cache performance and show the effects of varying cache parameters. By representing the factors that affect cache performance, we develop an analytical model that gives miss rates for a given trace as a function of cache-size, degree of associativity, block-size, multiprogramming level, task switch interval, and observation interval. The predicted values closely approximate the results of trace drive simulations while requiring only a small fraction of the computation cost.

Key Words and Phrases: Cache, cache model, interference, locality of references, measurement and analysis, multiprogramming, program behavior, run length; start up, trace driven simulation, working set.

1 Introduction

1.1 The case for the analytical cache model

Two methods predominate for cache analysis: trace driven simulation and hardware measurement. The survey article by Smith [21] uses the former technique extensively, while a comprehensive set of hardware measurements is presented by Clark [6]. Other examples of cache studies using the above methods include [3,5,9,11,24,20,23,1]. These techniques provide an accurate estimate of cache performance for the measured benchmarks. The problem is they cannot be used to obtain quick estimates or bounds on cache performance for a wide range of programs and cache configurations. Simulation is costly and must be repeated for each possible cache organization.[1] Large caches requiring longer traces for simulation exacerbate the problem. Multiprogramming effects are seldom studied using simulation due to the lack of multitasking traces; the availability of such traces [1] introduces additional dimensions over which simulations must be done. Hardware measurement, which usually involves costly instrumentation, requires an existing cache and gives data for only one cache organization (sometimes with possible size variations [6]). Furthermore, simulation and measurement inherently provide little insight into the nature of programs and the factors that affect cache performance. Analytical models on the other hand, if simple and tractable, can provide useful "first cut" estimates of cache performance. Simple models provide more intuition, but may lack the accuracy of more complex models. Therefore, if more detailed results are needed, simulations can be carried out to fine-tune the cache organization.

There are added advantages to having a simple model for cache behavior. For example, an understanding of the exact dependence of cache miss rate on program and workload parameters can identify those aspects of program behavior where effort would be best justified to improve cache performance. Cheriton et. al. [4] suggest that building caches with large blocks (cache pages), in conjunction with program restructuring to exploit the increased block sizes, could yield significant performance benefits. A cache model that incorporates the spatial locality of programs would be useful in analyzing the effects of various program restructuring methods. In addition, this model could be incorporated into optimizing compilers to evaluate tradeoffs in decisions, such as procedure merging, that affect cache performance. Also, in a multiprogramming environment, the tradeoff between higher resource utilization and degradation in cache performance with the level of multiprogramming can be easily assessed with a simple analytical cache model.

1.2 Overview of the model

Our cache model is hybrid in nature, involving a judicious combination of measurement and analytical techniques yielding efficient cache analysis with good accuracy. Since the intended application of our model is to obtain fast and reliable estimates of cache performance, the time spent in measurement of parameters from the address trace and, more importantly, in model computations must be significantly less than simulation. To minimize the number of parameters that have to be recorded and stored, average

[1] Stack processing techniques can sometimes be used to reduce the number of simulations [16].

quantities are often used instead of detailed distributions. This also decreases computation time. Besides, the bottom line of interest may be an average, for example, the miss rate. Mean Value Analysis (MVA), an example of this approach [15], gives accurate predictions of computer system performance using average measured system parameters to drive analytical models. MVA has been a key motivating factor in our cache modeling efforts.

Our model predicts cache miss rates for various cache organization and workload parameters. The miss rate is chosen because it is a key performance figure. Cache performance data for a given address trace is derived in two steps. First, we analyze the reference stream and record a few parameters assuming some basic cache organization, for example, a direct mapped cache of size 1K words and unit block-size. The basic organization can be chosen in the neighborhood of the cache type of interest to improve accuracy of the results. The model parameters, which are extracted from an address trace, are meaningful by themselves and provide a good indication of cache performance. In the second step, we vary cache and workload characteristics and project cache performance from the model. The model includes cache parameters such as cache-size, degree of associativity, and block-size; and workload characteristics such as multiprogramming level and the time between process switches. Cache miss rates can be derived by representing the following factors that cause cache misses:

1. **Start-up effects:** When a process begins execution for the first time on a processor, there is usually a flurry of misses corresponding to the process getting its initial working set into the cache. These first-time misses are due to start-up behavior. In the early portion of any trace, a significant proportion of the misses in a large cache can be attributed to start-up effects. This effect is also observed when a program abruptly changes phases of execution. Since miss-rate statistics for different phases of the same program are often widely uncorrelated, just as those of different programs are, each phase has to be separately characterized for maximum accuracy. However, for simplicity, or if the phases are short or if phase changes are small, their effects can be smoothed into the non-stationary category. Start-up effects are excluded when warm-start (or steady-state) miss rates are needed [8]. Inclusion yields cold-start miss rates.

2. **Non-stationary behavior:** This refers to the misses that occur when references are fetched for the first time after the start-up phase. Non-stationarity is a time dependent effect that occurs when a program changes its locality gradually. Any program performing the same sort of operation on each element of a large data set shows this behavior. Subtle changes of phase over small intervals of time, corresponding to change in cache working set, can also be modeled as a non-stationary effect. Often all phase behavior within a trace can be conveniently treated as non-stationary. While the overall miss rate might not be in error, the resulting transient miss rate, as predicted by the model, would show a smoother variation than in practice.

   Start-up and non-stationary behaviors are evident from working set plots [7] of programs. Working set plots have a roughly bilinear nature with a sharply rising initial portion corresponding to start-up, and a gradual slope thereafter denoting

non-stationarity.[2]

The misses caused by the above two effects are dependent on the block-size – assuming that a block is fetched from the main store on a cache miss. Increasing the block-size (up to a limit) takes advantage of the spatial locality in programs and reduces cache miss rate. A model of spatial locality is proposed to account for this effect.

3. **Intrinsic interference:** Due to finite cache size multiple memory references of a process may compete for a cache set and collide with each other. If the number of cache sets is S, then all addresses that have the same index (i.e., a common value for the address modulo S) will have to be in the same cache set. Interference misses occur when these references have to be fetched after being purged from the cache by a colliding reference of the same process. This effect depends both on how the addresses are distributed over the address space – a static effect, and the sequencing of references, a dynamic effect.

   The static characterization is based on the assumption that any reference has a uniform probability of falling into any cache set. The hashing operation (e.g., bit selection) that maps the large address space of programs to the much smaller cache space effectively randomizes the selection of any given cache set. Our results, and also those of Smith [22], show that this assumption is quite reasonable. The dynamic component is represented using a measured parameter from the trace called the collision rate. Unlike the distribution of blocks in the cache, which is dependent on the cache-size, the collision rate is shown to be reasonably stable over a wide range of cache and block sizes.

4. **Extrinsic interference:** Multiprogramming is an additional source of misses because memory references of a given process can invalidate cache locations that contained valid data of other processes. The impact of multitasking, not widely considered in previous cache studies, is particularly important for large caches [1] where a large fraction of the misses tend to be clustered near process switch points. Extrinsic interference will increase with the level of multiprogramming and decrease with the quantum of time that a process executes before being switched out. Other causes of extrinsic interference such as I/O and cache consistency invalidations are not included in this study but could be added.

   Extrinsic interference is modeled in the same manner as intrinsic interference with only the static characterization being necessary. The dynamic component, characterized by the collision rate in intrinsic interference, is not needed because a collision can happen only once per colliding reference after a process switch.

All the above effects will be considered in deriving a comprehensive model of cache behavior. Start-up effects and extrinsic interference characterize the transient behavior of programs, and non-stationary effects, intrinsic and extrinsic interference determine steady-state performance. The extent to which each effect contributes to the miss rate is a strong function of the cache organization and workload. In small caches misses are predominantly caused by intrinsic interference, whereas in caches large enough to

[2]Please see Figure 13 for an example working set plot.

hold the working set of the program, non-stationarity and extrinsic interference are the important sources of misses.

1.3 Related research

Our approach differs from some of the earlier memory hierarchy modeling efforts that tend to focus on some specific aspect of cache performance but did not adequately address the issue of a comprehensive cache model.

Some of the early memory hierarchy studies use empirical models. Chow [13] assumes a power function of the form $m = AC^B$ for the miss ratio, where C is the size of that level in the memory hierarchy, and A and B are constants. They do not give a basis for this model and do not validate this model against experimental results. Smith [21] shows that the above function approximates the miss ratio for a given set of results within certain ranges for appropriate choices of the constants. However, no claims are made for the validity of the power function for other workloads, architectures, or cache sizes.

The Independent Reference Model [2] is used by Rao to analyze cache performance [17]. This model was chosen primarily because it was analytically tractable. Miss rate estimates are provided for direct-mapped, fully-associative, and set-associative caches using the arithmetic and geometric distributions for page reference probabilities. A problem with this technique is that it assumes fixed page sizes and the number of parameters needed to describe the program is very large. Furthermore, validations against real program traces are not provided.

Smith focused on the effect of mapping algorithm and set-associativity [22] using two models, a mixed exponential and the inverse of Saltzer's linear paging model [19], for the miss ratio curve of a fully associative cache. The miss-rate formulas compared well with trace driven simulation results. However, separate characterization is necessary for each block-size, and time dependent effects and multiprogramming issues are not addressed.

Haikala [10] assessed the impact of the task switch interval on cache performance. He uses a simple Markov chain model to estimate the effect of cache flushes. The LRU stack model of program behavior [25] and geometrically distributed lengths of task switch intervals are assumed. The model is reasonably accurate for small caches where task switching flushes the cache completely and pessimistic for large caches where significant data retention occurs across task switches [1].

Strecker [27] analyzes transient behavior of cache memories for programs in an interrupted execution environment using the linear paging model for the miss ratio function. The analysis accounts for data retention in the cache across interruptions. The form of the miss ratio function used is $(a + bn)/(a + n)$, where n is the number of cache locations filled; a and b are constants obtained by measuring the program miss rates at two cache sizes. Predicted miss rates of several real programs run individually and interleaved for various execution intervals compare reasonably well with trace driven simulation results.

The transient behavior of caches is also studied by Stone and Thiebaut [26]. They

calculate the minimum number of transient cache refills necessary at process switch points as a function of the number of distinct cache entries touched by the program, also called the program footprint, for two processes executing in a round robin fashion. However, they do not give a method of obtaining the footprint and validation of their results is also not provided.

The effect of block-size was not included in the above studies; program behavior had to be separately characterized for each block-size. Kumar [14] investigates spatial locality in programs. Kumar proposes an empirical technique to estimate the working set of a program for different block sizes. The miss rate calculated as a function of block-size is shown to correlate well with the results of trace driven simulation. Besides being specific to block size effects, the study has certain drawbacks. Validation is carried out only for very large caches to exclude the effect of program blocks colliding with each other in the cache. Hence, only start-up effects can be considered to be adequately modeled in this study.

The above papers have been valuable milestones in analytical cache modeling and have characterized various aspects of cache behavior. However, the program models used and the assumptions made in most studies tailored the analysis to some specific cache parameters limiting the scope of their application.

In the following section, we first describe a basic cache model taking into account start-up, non-stationary, and interference effects for a direct mapped cache with a fixed block-size. Section 3 extends the model by including the effect of set-size, block-size and multitasking. A discussion of the results of our experiments and model validations against several address traces are provided in section 4. Section 5 summarizes the model and our results.

2 A basic cache model

This section describes a model for direct-mapped caches with a fixed block-size. The total number of misses is calculated as the sum of the misses due to start-up, non-stationary, and intrinsic interference effects. Only one process, i, is assumed to be active. In general, all parameters associated with this process will be subscripted with the letter i. However, for simplicity, we will bring in this subscript only when necessary to distinguish between processes.

A notion of time in the context of a reference stream is necessary to study the transient behavior of caches. We assume that each reference represents a time step. We also define a larger time unit called a *time granule (tg)* for use in the model. A time granule is sequence of τ references. Average parameter values are calculated over a time granule. Processes are assumed to execute on the processor for an integral multiple of time granules and switch on a *time granule* boundary. This may not be the case in real life, but, as we shall see, this approximation is still useful in predicting average cache performance.

Cache organizations, C, are denoted as a triple (S, D, B), where S in the number of sets, D is the set-size or degree of associativity, and B is the block-size or line-size in four byte words. For a detailed description of the cache terminology used in this paper please refer to Smith's survey paper [21]. In other studies S has been called number of rows, D the number of columns, and B the cache page-size. The cache-size, in terms of blocks, is the product of the number of sets and the set-size. It is assumed that a block of data is transferred between the main store and the cache on a cache miss.

We have tried to validate the basic model at every step against simulation results. For intermediate results, we use an address trace of Interconnect Verify (IV) of length 400,000 references. Interconnect Verify is a program used at Digital Equipment Corporation's Hudson site to compare two connection net lists. Our trace is a sample of IV operating on a cache chip.

First we need some definitions:

τ: The number of references per time granule. A typical value is 10,000. As we shall see in the discussion on sensitivity analysis in the appendix the choice of τ is not critical to the analysis. Ideally, τ should be bounded below by the average time spent in start-up activity[3].

T: Total number of time granules of a process i, or the trace length in number of references divided by τ.

t: Represents a particular time granule. t varies from one to T.

$u(B)$: The average number of unique memory blocks accessed in a time granule by the process. Clearly u is a function of block-size. u is similar to the working set with parameter τ as defined by Denning [7].

[3] A definition for the start-up period is provided in Appendix A

$U(B)$: The total number of unique memory blocks used in the entire trace. In the basic cache model, we drop the use of B and use the notation U since the block-size is kept constant. In practice, U is less than $T * u$ because many references are common across time granules.

$m(C,t)$: The miss rate for process i up to and including its t^{th} time granule for cache $C : (S,D,B)$. This is the total number of misses divided by the total number of references in the first t time granules.

2.1 Start-up effects:

The initial filling of the cache causes start-up misses. Let us assume for a moment that all these misses happen in the first time granule. Note that time granule size is usually chosen to be large enough to account for most of the start-up activity. Then for the first time granule, the miss-ratio due to start-up effects is the ratio of the number of unique references accessed in that time granule to the total number of references:

$$m(C,1)_{startup} = \frac{u}{\tau}$$

The start-up component decreases monotonically with time because the number of start-up misses is constant:

$$m(C,t)_{startup} = \frac{u}{\tau t} \tag{1}$$

The above formula shows that even if the cache filling takes more than one time granule, the miss rate will be in error only as long as cache filling due to start-up takes place. This term becomes vanishingly small for large traces. For Interconnect Verify, with a block-size of one word, and τ chosen to be 10000, the average number of unique memory blocks accessed in a time granule, $u(1)$, is 1624.

2.2 Non-stationary effects:

Equation 1 considered the misses that occur in fetching blocks for the first time during the initial time granule and did not take into account the fact that in each time granule the process could be renewing parts of its working set. For example, this behavior is seen when a program is operating at any instant on a small segment of a large array of data. As the program moves to new portions of the array, these new references will give rise to an added component of misses.

In our model, the first time granule has a large number of new blocks, which is the initial working set u. In subsequent periods, only a fraction of these will be renewed. Let f_u be the fraction of unique references, u, that are new in every time granule. In keeping with our assumption of the average being a good predictor of the actual value, we can estimate the total number of non-stationary misses as the number of unique blocks in the entire trace minus the initial start-up misses. On dividing by T, the total number of time granules, the average number of non-stationary misses in a time granule is obtained. The ratio of this quantity to u gives f_u:

$$f_u = \frac{U - u}{T\ u}$$

where, U are the number of unique blocks in the entire trace and T are the total number of time granules represented by the trace. Thus the cumulative number of misses up to the t^{th} time granule due to non-stationary effects, is

$$f_u \, u\,(t-1)$$

and the corresponding miss rate is obtained by dividing the number of non-stationary misses by the number of references, $t\tau$,

$$m(C,t)_{nonstationary} = \frac{u}{\tau} \frac{f_u\,(t-1)}{t} \tag{2}$$

Interconnect Verify, with a block-size of one word, has, $u(1) = 1634$, $U(1) = 7234$, $T = 40$, and $f_u = (7234-1624)/(40*1624) = 0.0863$. Note that the non-stationary component of misses over the entire trace is over four times the start-up component.

2.3 Intrinsic Interference:

Some cache misses are caused by multiple program references competing for the same cache block. This effect lessens as cache-size grows larger because fewer references on the average compete for any given cache block. To include this effect in the model, we need to estimate the number of collision induced misses that will occur as a function of cache-size. In general, the set-size or associativity plays an important role, but for the present, the analysis assumes a direct-mapped cache. As mentioned before, estimation of interference misses involves both a static and a dynamic factor.

2.3.1 Static Characterization

For the static characterization, the distribution of program blocks into cache sets will be modeled. From this model, the number of blocks that collide with each other will be estimated. We define a *collision set* as a set that has more than one block mapped to it, and, a *colliding block* (or an interfering block) any block that maps into a colliding set. In other words, a colliding block is one that could be potentially displaced from the cache by another block.

The key assumption we make in our derivations is that a program block has a uniform probability of being in any cache set. Many factors contribute to the validity of this assumption. First, the hashing operation that maps the large process address space to the smaller number of sets in the cache randomizes the selection of any given set. This is not entirely true (especially for small block-sizes and simple hashing functions such as bit selection) because reference streams are highly sequential in practice, and given that a set has been recently occupied, the probability that an adjacent set will be accessed is higher that average. This factor causes estimated miss rates to be generally higher than actual. This is less of an issue in physical caches because virtual pages, often sequential, can be mapped to an arbitrary physical page; but the correlation within a page is still high. In virtual caches, the Process Identifier (PID) is sometimes hashed in with the address used to index into the cache causing references of different processes to be uncorrelated with each other. Also, since code and data for user and

system tend to reside in different segments of the address space, they are mutually independent with respect to the cache sets they occupy.

Assuming random placement in the cache, the probability, $P(d)$, that d blocks fall into any given cache set (depth of overlap d) has the binomial form. As usual we leave out the dependence of this probability on B for notational brevity.

$$P(d) = \binom{u}{d} \left(\frac{1}{S}\right)^d \left(1 - \frac{1}{S}\right)^{u-d} \tag{3}$$

Note that the probability that a block maps into a given cache set is $1/S$, and that d blocks map into this cache set is $(1/S)^d$; $(1 - 1/S)$ is the probability that a block does not map into the given set.

To validate this model, the number of blocks that map into a set at a depth of overlap d is calculated for Interconnect Verify. The correspondence between calculated and observed values (Figure 1) for a wide range of cache-sizes (1K to 64K bytes) is very close. Considering depths of up to seven, which includes 99% of the references, the mean and maximum errors in estimation are less than 15% and 42% respectively for a $4K$ byte cache. There are two caveats to this error presentation. First, in our mean value modeling approach, only the mean error is important, and second, noting that errors in the tail of the function do not significantly affect the predictive capability of the model, a better indication of performance is provided by focusing on depths less than five which corresponds to 95% of the references. The mean and maximum errors in this range are 2% and 4% respectively. The component of error in the intrinsic interference dominated miss rate attributable to inaccurate calculation of the number of overlapping blocks for the 4K byte cache is only 2.6%, which further corroborates our argument.

The number of interfering blocks can be calculated straightforwardly from Equation 3. Collisions will not occur in any cache set that has at most one program block. Hence, in a direct-mapped cache, the probability that a cache set has no colliding blocks is $P(0) + P(1)$, and the number of such non-colliding blocks is $S * [0 * P(0) + 1 * P(1)]$, which is $SP(1)$. In general, for any number of sets S, we can compute the number of interfering blocks to be the difference between the number of unique memory blocks and the number of blocks that do not collide, or,

$$Number\ of\ interfering\ blocks = u - S\ P(1) \tag{4}$$

Figure 2 shows colliding blocks as a function of the number of sets. For small caches, all the blocks collide yielding a maximum of u colliding blocks, and for large caches this number approaches zero. Again the correspondence between predicted and actual values is quite good with average the error 14% and the maximum error 72%. The large error in $64K$ sets is of little consequence because the miss rate is predominantly start-up dominated; the impact of incorrectly estimating the number of colliding blocks on the miss rate for a cache with 64K sets is less than 9%. The mean and maximum errors excluding 64K sets are 8% and 33% respectively.

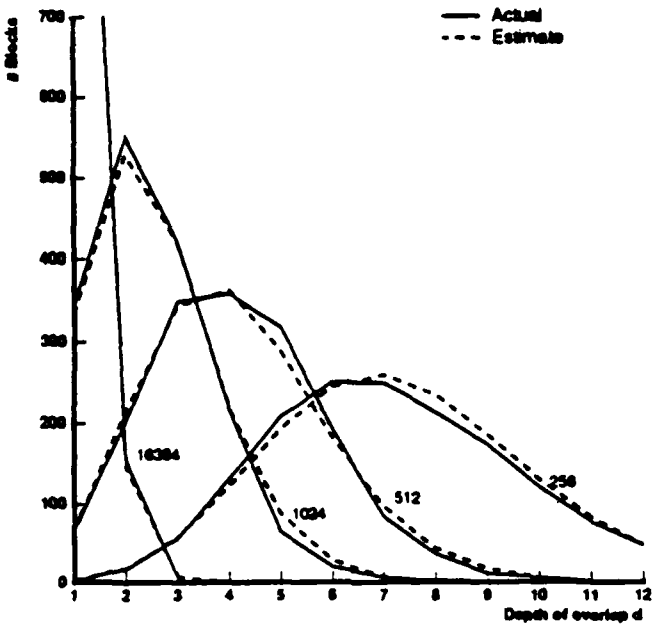


Figure 1: Number of blocks of size one word (4 bytes) that map into a cache set with depth of overlap *d*. Values on the figure indicate cache-size in number of sets.

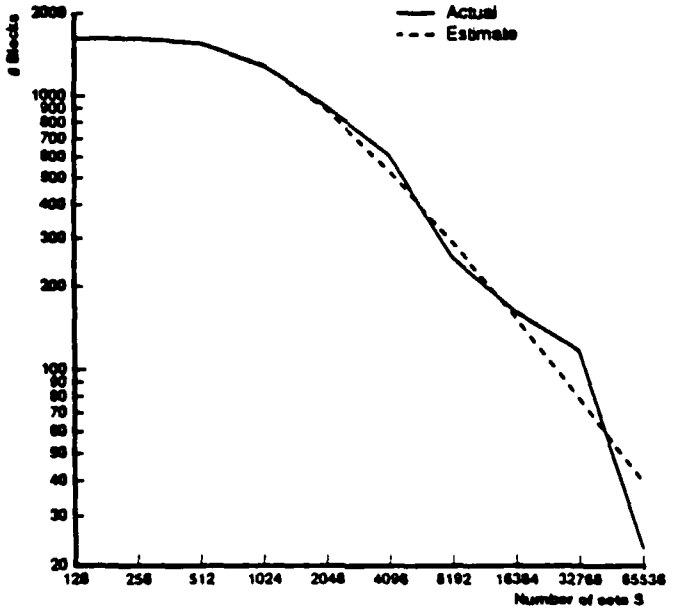


Figure 2: Number of collision blocks as a function of the number of sets.

2.3.2 Dynamic Characterization

The above formula gives just the number of colliding blocks, or the static portion; it does not indicate when colliding blocks actually induce misses, nor does it tell us anything about the number of times that the blocks actually purge each other from the cache (the dynamic component), which is what we need to compute the miss rate. First, we will derive some loose bounds on this number and on the miss-rate component due to intrinsic interference, then present a more accurate characterization of the miss rate.

We define a *dead block* to be one that is not going to be referenced in the future. A *live block*, on the other hand, is still active. The minimum number of misses occur if every colliding block that is brought into the cache lands on top of a *dead* one. For example, if two variables that compete for a cache block are live only in alternate halves of a time granule the number of collision induced misses due to these two blocks will be zero. However, if colliding references are very finely interleaved, then these will cause thrashing in that cache block. Thus, a miss is induced by a collision only if the displaced block is live. In our model, the average number of times a reference is repeated per time granule is τ/u giving rise to a possible maximum of τ/u intrinsic interference misses per colliding reference.

Let the *collision rate*, c, be the average number of times a live colliding block in the cache is purged due to a collision in a time granule. In other words:

$$c = \frac{Number\ of\ times\ live\ variables\ are\ purged\ in\ a\ direct\ mapped\ cache\ per\ tg}{Number\ of\ colliding\ references} \tag{5}$$

The actual miss rate due to intrinsic interference is given by,

$$\begin{aligned} M(C,t)_{intrinsic} &= \frac{c * Number\ of\ colliding\ references}{\tau} \\ &= \frac{c[u - S\ P(1)]}{\tau} \end{aligned} \tag{6}$$

We now need a characterization for the collision rate, c, which is bounded as

$$0 \leq c \leq \frac{\tau}{u}$$

The high value is attained when the cache has only one set and every reference is a miss. The collision rate, c, will depend on a number of factors including the number of times loops are executed in the given program and the time interval between the live periods of colliding blocks. Intuition leads us to believe that c will be approximately the same for different cache sizes for any given program up to the point where the cache becomes small enough that all cache sets are filled. Beyond this ceiling point, c will monotonically increase until it reaches its maximum, τ/u, for $S = 1$. Our experiments (presented in Appendix A) show this to be true. Thus, we can measure a value for c from the given trace for a representative direct-mapped cache with a block-size of one

and number of sets S_0. We will use this value of c in miss-rate projections for most other cache sizes and organizations as well.[4] Hence, we derive c as follows:

$$c = \frac{Number\ of\ times\ live\ variables\ that\ are\ purged\ in\ a\ tg\ when\ S = S_0}{Number\ of\ colliding\ references}$$

For our example we choose $S_0 = 1024$. Total number of live references purged in Interconnect Verify per time granule is 2391; the probability that only one block maps into a given cache set, $P(1) = 0.32$; number of unique references in a time granule, $u = 1624$; and the number of colliding references $= 1624 - 1024 * P(1) = 1291$. The collision rate

$$c = \frac{2391}{1291} = 1.9$$

We need to verify whether c is stable over different numbers of sets. Table 1 compares calculated and actual values of c for Interconnect Verify, and we see that c is reasonably stable and the variations do not seem to show any pattern.

No. sets	c measured	c calculated
1K	1.9	1.9
2K	2.0	1.9
4K	2.1	1.9
8K	1.6	1.9
16K	1.9	1.9
32K	2.2	1.9
64K	1.2	1.9

Table 1: Collision rate for various cache-sizes.

Summarizing, the basic cache model for direct-mapped caches gives the miss rate as a sum of start-up (Equation 1), non-stationary (Equation 2), and intrinsic interference effects (Equation 6):

$$m(C,t) = \frac{u}{\tau t}[1 + f_u(t-1)] + \frac{c}{\tau}[u - SP(1)] \qquad (7)$$

In steady-state (t tends to ∞) the miss rate becomes independent of start-up effects:

$$\frac{u\,f_u}{\tau} + \frac{c}{\tau}[u - SP(1)].$$

This simple cache model predicts miss rates for direct-mapped caches with a fixed block-size both as a function of time and cache-size. Figures 3 and 4 compare the results of the simple model with trace driven simulation results using the benchmark Interconnect Verify. Further validations against other traces are provided after the following section. The IV trace has 400,000 references. Block-size is chosen to be 1 word. S_0 for parameter extraction was 1K.

[4]The exceptions are discussed in the Appendix.

Miss rates for various cache-sizes as a function of time granules are in Figure 3. Prediction is quite accurate for caches of size 1K sets (mean error 6%, maximum error 30%), 16K (17% and 30%), and 64K sets (13% and 35%). In the 4K set cache (mean error 25%, maximum error 28%) the estimated values are lower than the actual values because both c and the number of colliding blocks are underestimated. Miss rate is substantially overestimated for small caches in the first time granule ($1K$ and $4K$ caches in Figure 3) because the model assumes that intrinsic interference is uniform over time. In practice, the initial portion of a trace has a relatively few intrinsic interference misses since most of the misses are attributable to start-up activity. Prediction suffers more in smaller caches because the intrinsic interference miss-rate component is higher. This problem is easily fixed by only including intrinsic interference misses from the second time granule onwards. Prediction for the 64K set cache is quite good despite c being off by almost 60% because the number of colliding blocks is so small as to make intrinsic interference an insignificant component in the miss rate. Interconnect Verify shows a subtle phase change near the 20th time granule, which the model cannot detect because of its averaging property.

Figure 4 shows miss rates as a function of the number of sets. Estimated miss rates are very close to actual for caches ranging from 128 sets to 64K sets. The predicted curve yields a good overall fit to the actual curve even though individual miss rates may be in error. Differences are usually attributable to erroneous estimates of the collision rate c.

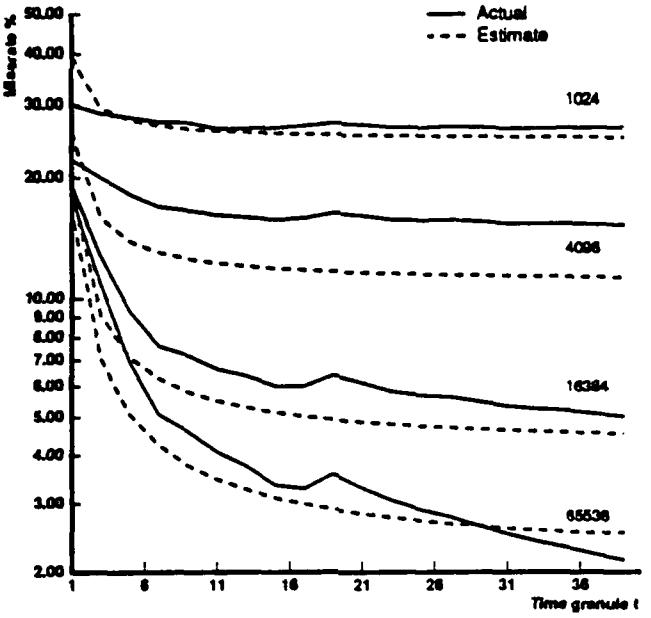


Figure 3: Miss rate vs. time granule for caches with 1024, 4096, 16384, and 65536 sets. Actual values represent simulated miss rates.

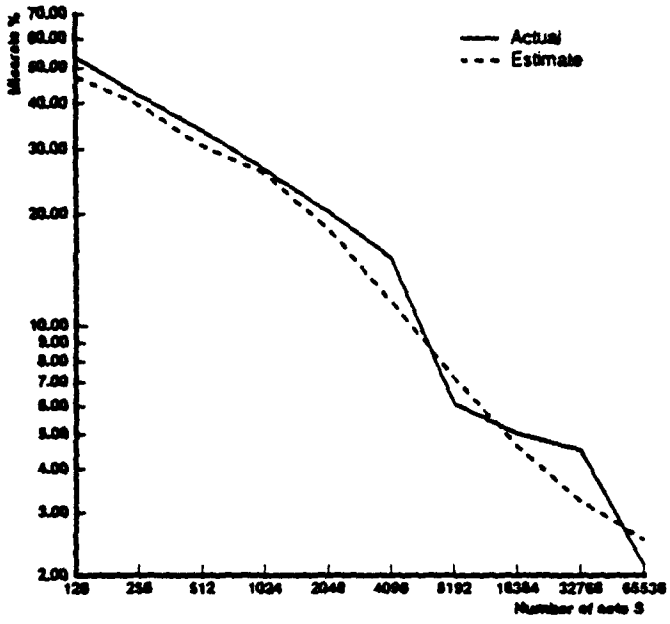


Figure 4: Miss rate vs. number of sets S.

3 A comprehensive cache model

In the previous section we derived miss-rate estimates for a single process using a direct-mapped cache and a given block-size. We now extend the model to include set-size (degree of associativity), block-size, and multiprogramming.

3.1 Set-size

The intrinsic interference term is affected when set-size is changed. If the set-size is D, only those cache sets with more than D competing blocks will have collisions. Thus, the number of colliding blocks decreases monotonically with increasing degree of associativity D when the number of sets, S, is constant. The effect of set-size on the collision rate, c, is more subtle. Recollecting that in a collision set a miss is induced if a live block is purged, more blocks can co-exist in a set without purging each other if the set-size is increased, thus decreasing c. Clearly, both the static and dynamic components that comprise intrinsic interference depend on set-size. As before, we first derive the static component, followed by the dynamic part.

3.1.1 Static characterization

To estimate the number of colliding blocks for any set-size D, we start with the probability, $P(d)$, that d blocks fall into any given cache block as in Equation 3.

$$P(d) = \binom{u}{d} \left(\frac{1}{S}\right)^d \left(1 - \frac{1}{S}\right)^{u-d}$$

The number of sets that have d competing blocks is therefore $S\ P(d)$. The number of blocks that collide, for a cache with degree of associativity, D, is obtained by subtracting the total number of blocks that overlap with depth less than D from the total number u:

$$u - \sum_{d=0}^{d=D} S\ d\ P(d)$$

Figure 5 shows the number of colliding blocks in the Interconnect Verify trace for caches with set-sizes one through 16. The number of colliding blocks decreases with set-size for most caches. For the $1K$ set case, however, the converse is true. This can be explained as follows: Consider a cache with S sets and set-size D. Let set s_1 of this cache have $d_- < D$ blocks mapped into it, and set s_2 have $d_+ > D$. Further, let the positions occupied by the two sets in the cache be $S/2$ apart. Of the two, by our definition, only s_2 is a collision set and the number of colliding blocks contributed by s_1 and s_2 is d_+. Now, keeping the cache-size the same the set-size is doubled. Because the number of sets is halved, the blocks in the two sets (totaling $d_- + d_+$) will land into the same cache set. It is easy to see that the number of colliding blocks will be less than before if $d_- + d_+ <= 2D$ and greater if $d_- + d_+ > 2D$. Because the latter case is often possible for small caches, the number of colliding blocks can actually increase

with greater set-size.[5] As we shall confirm later, this behavior can reduce the advantage of associativity for small caches.

The correspondence between observed and predicted values in Figure 5 is good. The mean and maximum errors for a $1K$ set cache are 1% and 2% respectively; for a $2K$ set cache 5% and 16% respectively; and 38% and 88% respectively for a cache with $4K$ sets. As stated before, the tail portion of the curves can be off by a large relative amount and do not contribute significantly to the final result. In this case, the relative error in our colliding block estimate for set-size 16 is unimportant because the miss rate is start-up dominated; and in fact, the corresponding errors in miss rate for a $4K$ set cache is less than 5%. Hence, the mean and maximum errors for a cache with $4K$ sets not including set-size 16 become 25% and 33% respectively.

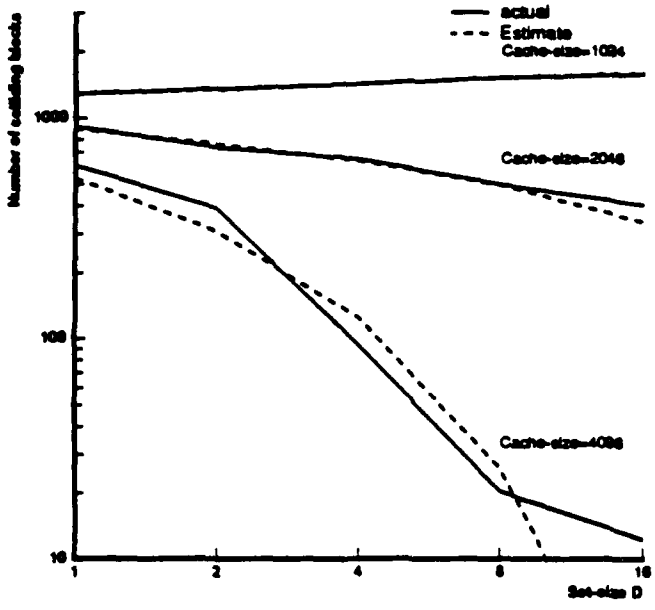


Figure 5: Number of colliding blocks vs. D. Cache-size is specified in 4 byte words.

3.1.2 Dynamic characterization

We need the collision rate, or the average number of times a block collides, in addition to the static factor. For caches with set-size greater than one, the *replacement algorithm* has to be considered in determining the average number of times a block collides. Since we do not have use information, modeling LRU replacement is hard. Appendix B shows how $c(D)$ can be obtained for set-sizes greater than one from the value of c measured from a direct-mapped cache (Please see Equations 5 and B.2). This method approximates random replacement. The results can be used as a good estimate for the miss rates given by other non-usage based replacement schemes such as FIFO [21].

[5]The miss rate, however, deteriorates only if the collision rate, c, does not correspondingly decrease.

Smith also shows that random replacement has about ten percent worse cache miss rate than LRU replacement on average. Thus, our predicted miss rates can be used as a loose estimate on those with LRU replacement also.

The total number of collisions in a time granule due to intrinsic interference is the number of colliding blocks weighted by the collision rate, $c(D)$. The aggregate number of misses per time granule due to intrinsic interference in a cache with S sets and set-size D is therefore

$$c(D)\left[u(B) - \sum_{d=0}^{d=D} S\ d\ P(d)\right] \tag{8}$$

and the corresponding intrinsic miss rate component is

$$m(C,D)_{intrinsic} = \frac{c(D)}{\tau}\left[u(B) - \sum_{d=0}^{d=D} S\ d\ P(d)\right]. \tag{9}$$

3.2 Modeling spatial locality and the effect of block-size

The derivations in the previous sections assume that the block-size is kept at some constant value throughout all measurements and analysis. We now consider the effects of changing block-size. By increasing the block-size, we can take advantage of the spatial locality in address streams because the number of unique blocks is a decreasing function of the block-size. The dependence of u and U on the block-size is determined by two factors: the distribution of run lengths and the distribution of space interval between runs.

The *distribution of run lengths* is needed to estimate the effect of changing block-size, where a *run* is defined as a maximum stream of sequential references. As an example, if some data objects are isolated words, then we will need one block for each of these data items for most reasonable block-sizes. Runs of length B aligned on block boundaries will be contained in blocks of size B. Further, if blocks cannot be aligned on arbitrary word boundaries then the alignment of the given run within a block will also matter.

The second factor accounts for the capture of multiple runs by a single block. While run lengths usually range from one to ten words, empirical cache studies have shown that block-sizes far in excess of ten words can still capture additional localities. This effect can be explained by the fact that a large enough block can capture more than a single run. Thus, we need *the distribution of space intervals between runs* to predict the usefulness of increasing block-sizes beyond average run lengths.

The ensuing discussion uses the distribution run-lengths to characterize spatial locality in programs. From this characterization the miss rate dependence on block-size is determined. Appendix C extends the discussion by considering the effect of capturing multiple runs in a single block.

3.2.1 Run Length Distribution

A direct method of obtaining run-length distributions is to measure it from the address trace of the program. Unfortunately, direct measurement encounters two problems.

First, typical address traces contain interleaved streams of instruction and data addresses. Even after separating these streams, the data addresses from the stack and the heap could be interleaved. For example, a VAX trace can contain a sequence starting with an instruction address, followed by addresses of the first, second, and third operands. Sequentiality of this nature[6] has to be detected in the address traces to derive run length statistics. An efficient method (albeit approximate) of separating these streams and identifying runs is to sort the references in successive segments of the trace.

The second problem is that an accurate characterization of run-lengths mandates the use of a large number of parameters – one for every possible run-length. To keep computation and the number of parameters to a minimum, we propose a simple Markov model to depict the spatial locality in programs. Then, an approximate distribution of run-lengths will be derived from two average parameters measured from the address trace.

In general, an n stage Markov model is required (Figure 6(a)) to characterize run-lengths, where n is the largest run-length in the trace. In the figure, state $R1$ corresponds to the beginning of a run; state Rk is reached if the first k addresses are sequential. f_{lk} is the probability that the next address will be sequential given that the first k addresses are. Since the longest run is of length n, the probability of a sequential reference in state Rn is zero, and a new run is begun with probability one.

In practice, the complication of a n stage model is not actually necessary; we can make a good approximation using a two stage model. The reason is quite simple. Addresses typically fall into two categories: those that form part of a run of unit length and those that do not. The former are called singular references. Addresses (in particular data) have a reasonably high probability $(1 - f_{l1})$ of being a singular address[7]. Further, given that an address is non-singular, it has a high probability of having a sequential successor (f_{li}, $i = 1, 2, 3, \ldots, n$). Instruction streams, and clustered accesses in data structures display this behavior. We have observed that non-singular addresses show a memoryless property to some extent. I.e., the probability that a non-singular address has a sequential successor does not depend strongly on the number of preceding addresses in the run.

The values of f_{lk} for Interconnect verify are: $f_{l1} = 0.56$, $f_{l2} = 0.72$, $f_{l3} = 0.70$, $f_{l4} = 0.84$, $f_{l5} = 0.86$, $f_{l6} = 0.90$, and $f_{l7} = 0.8$. The values of f_{lk} for $k > 1$ are quite close. This confirms that we can approximate the n stage Markov model by a two stage model as in Figure 6(b). f_{l1} remains the same. f_{l2} is chosen to be the weighted average of the other f_{lk}s. For Interconnect Verify, $f_{l1} = 0.56$ and $f_{l2} = 0.85$.

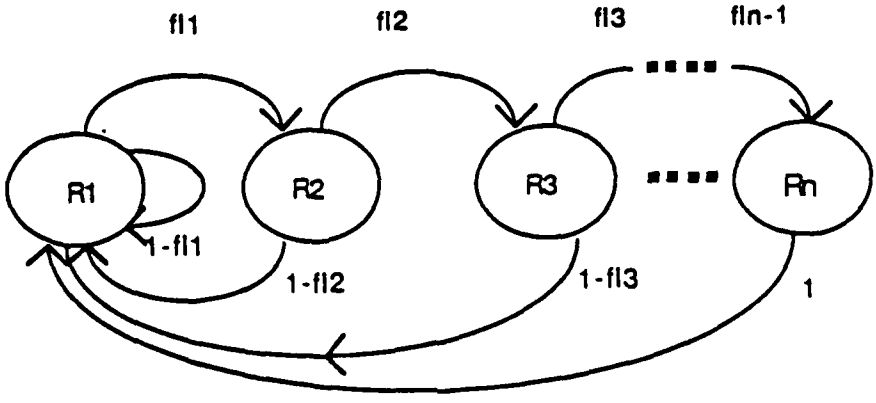
The probability of any run being of length l is given by,

$$\begin{aligned} R(l) &= 1 - f_{l1}, \quad l = 1, \\ R(l) &= f_{l1}\, f_{l2}^{l-2}\,(1 - f_{l2}), \quad l > 1. \end{aligned}$$

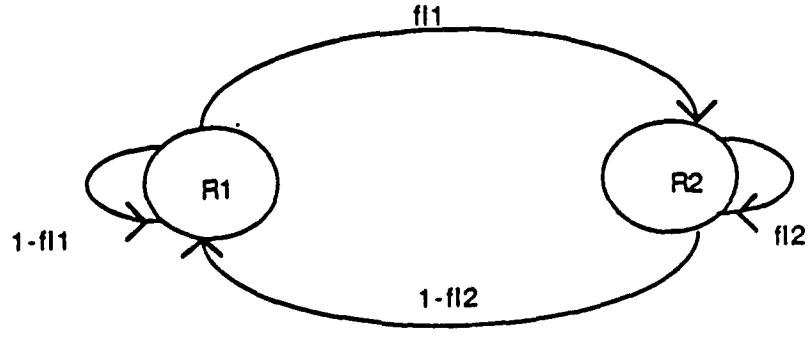
Furthermore, the probability of being in state R_1 in the steady state [28,12] is the fraction of references that begin a new run, the reciprocal of which gives the average

[6] Rau presents an interesting discussion on this issue [18].

[7] This is the reason why a two stage model is chosen; a one stage model would fail to capture the dichotomous nature of reference patterns.



(a) An n stage Markov model



(b) A two stage Markov model

Figure 6: Markov models depicting spatial locality in programs.

run length:

$$l_{av} = \frac{1 + f_{l1} - f_{l2}}{1 - f_{l2}} \tag{10}$$

and the number of unique runs is the number of unique references divided by the average run-length, that is $u(1)/l_{av}$. Figure 7 shows the distribution of run-lengths for Interconnect Verify. The dotted line is the approximation using the above simple two stage model.

The next step is to calculate $u(B)$, the number of unique blocks contained in the trace. We define the *cover* for a run to be the set of blocks that have to be fetched on average to bring the entire run into the cache. Note that a reference to any word in a block causes the whole block to be fetched. We start by assuming that a cover for a run can contain at most one run. Then, for a run of length l with ideal alignment (the run starts on a block boundary), at least $\lceil l/B \rceil$ blocks are needed. In general, the alignment is random and the average number of blocks needed to contain a run of length l, or the *cover-size*, is given by the following equation:

$$Cover\ size\ =\ 1 + \frac{l-1}{B} \tag{11}$$

For example, assuming a block-size of four words, exactly one block is needed for a run of length one. For a run of length two we need at least one block. Two blocks are used up if the run is aligned such that it crosses block boundaries. This happens with probability 1/4. So, the cover-size, or the average number of blocks allocated to the run is 1.25. Lacking the inclusion of multiple runs in a cover, each cover will waste a fraction of a block given by the following formula:

$$\left(1 + \frac{l-1}{B}\right) - \frac{l}{B} = \frac{B-1}{B}$$

In the above example three words are unused.

Hence, the total number of unique program blocks for a given block-size B corresponding to $u(1)$ unique words is the average number of blocks needed to cover a run of length l times the number of runs of length l, summed over all l:

$$u(B) = \frac{u(1)}{l_{av}} \sum_{l=1}^{l=\infty} R(l) \left(1 + \frac{l-1}{B}\right) \tag{12}$$

Figure 8 compares predicted and actual values of $u(B)$ as a function of block-size for Interconnect Verify. The mean error in the estimated values is 33% and the maximum is 115%. However, the mean and maximum errors are less than 5% and 8% for block-sizes up to 8 words. Thus, the above simple model is sufficient for block-sizes less than about 8 words. For greater block-sizes, the model incorrectly predicts that $u(B)$ is insensitive to block-size. The cause of this error lies in our assumption that a block can capture only a single run. Because average run lengths are of the order of four, larger blocks will be mostly empty having little impact on $u(B)$ in the model.

Actually, the probability of a block capturing more than a single run is non-negligible if the distance between the runs (inter-run interval) is small, as is often

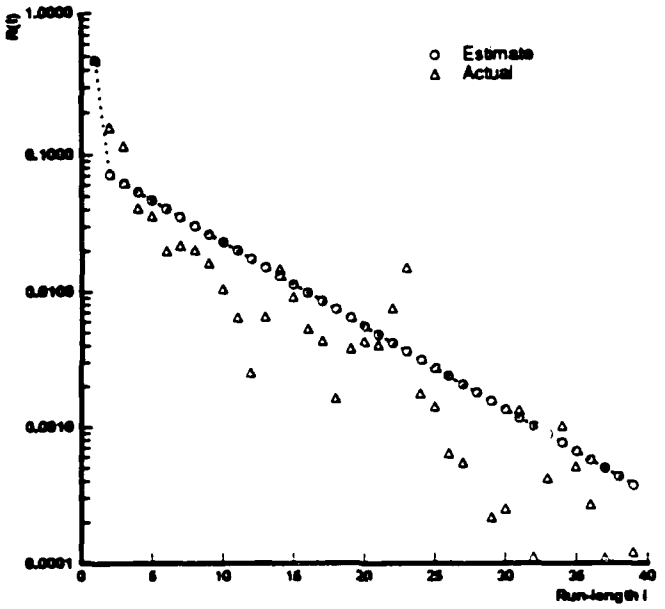


Figure 7: Distribution of run-lengths for the benchmark IV. $R(l)$ is the probability that a run is of length l.

the case, and it becomes necessary to include inter-run intervals in our discussion. For a detailed characterization of spatial locality please refer to Appendix C. Basically, the cover size for a given run, as calculated in Equation 11, includes a fraction of a block that could presumably cover neighboring runs. The portion of the block utilized for other runs, determined from a measured average inter-run interval and the run-length distribution, is subtracted from the cover size to yield the actual number of words allocated to the run. $u(B)$ is then computed as in Equation 12.

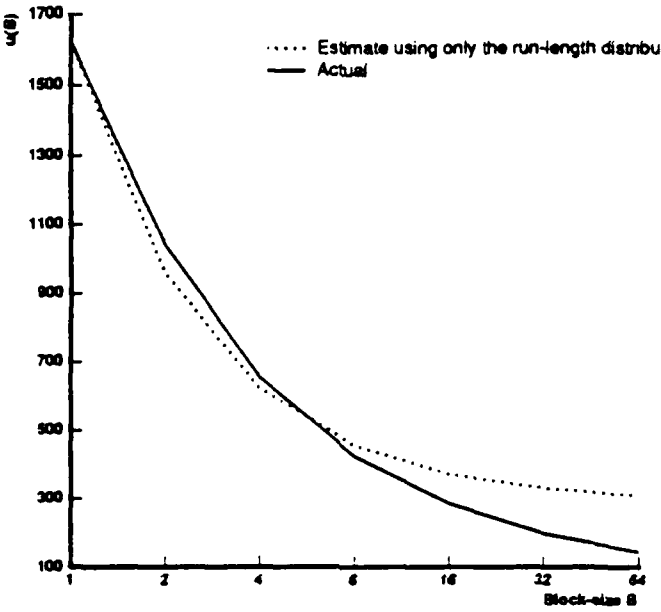


Figure 8: Average number of unique blocks, $u(B)$ in a time granule, τ, vs. the block-size B.

The miss rate can now be estimated using $u(B)$ and $U(B)$ calculated as shown above. The relevant miss rate formulation (as in Equation 7) is repeated here including the dependence on the block-size B.

$$m(C,t) = \frac{u(B)}{\tau t}\left[1 + f_{i,u}(B)\,(t-1)\right] + \frac{c(D)}{\tau}\left[u(B) - \sum_{d=0}^{d=D} S\; d\; P(B,d)\right] \qquad (13)$$

Appendix B provides a discussion on the impact of block-size on the collision rate c. Basically, the collision rate is not significantly affected by a change in the block-size for most cache sizes. Note that the collision rate reflects the dynamic sequence of referencing to various program localities (blocks), and a locality can be expected to have a similar dynamic behavior as its component words.

3.3 Multiprogramming

The discussion thus far assumed single process execution. Because workloads of real systems usually consist of multiple processes, and single process models tend to be optimistic, multiprogrammed cache models are necessary to accurately depict cache performance. Large caches can often hold the entire working set of a single process. In this case most misses occur immediately following a process switch when a process needs to refill its data purged by intervening processes.

The following discussion assumes round robin scheduling with constant duration time slices; in general, time slice lengths can be measured, or picked from an appropriate distribution with the constraint that each time slice equals an integral multiple of time granules. Let mpl represent the multiprogramming level and t_s represent the number of time granules in a time slice. Our derivation assumes that the cache is physically addressed, or in the case of a virtually addressed cache assumes that each process has a unique process identifier (PID) that is appended to the cache tags to distinguish between the virtual addresses of different processes. Flushing the cache on every process switch can also be modeled in an even simpler manner. We concentrate on the miss rate for a process i. Let $m_i(C, t_i)$ be the aggregate miss rate for process i after its t_i^{th} time granule, and, as before, let the average number of blocks of process i used in a time granule be u_i.

Let *carry over set* denote the set of references that a process leaves behind in a cache on being switched out and re-uses on its return, and $v_i(B)$ be the average number of blocks in the carry over set. The notion of a carry over set yields an accurate characterization of the transient nature of cache misses due to multiprogramming. It is easy to see that the maximum number of misses that can occur after a process resumes execution after being switched out is $v_i(B)$. The number of blocks in the carry over set is bounded above both by cache-size and program working set size. Thus, in a small cache multiprogramming induced misses are smaller in number than in a large cache.

Using the notion of a carry over set the effect of multiprogramming on cache performance is computed as follows. Suppose that a process i has been scheduled to run on the processor after being switched out. There would have been $mpl - 1$ processes between instances of process i. Due to these $mpl - 1$ intervening processes some fraction, $f_{i,v}(mpl)$, of the blocks in its carry over set gets purged. The resulting additional misses will be equal to the product of $f_{i,v}(mpl)$ and the number of blocks in the carry over set. The number of blocks in the carry over set in a cache with number of sets S, set-size D, and block-size B, is approximately given by the sum of all the blocks in cache sets that do not collide:

$$v_i(B) = \sum_{d=0}^{d=D} S\ d\ P_i(d). \tag{14}$$

This is not strictly true because we have excluded cache entries that could potentially be purged due to intrinsic interference. The implicit assumption is that a colliding variable is more likely to be purged due to intrinsic interference than due to the intervening processes. This makes the miss rate estimates slightly optimistic. $v_i(B)$ could also be computed by summing all cache resident blocks of process i to give pessimistic miss rate estimates. To arrive at a more accurate estimate of the carry over set, a fraction,

equal to the ratio of the size of the time granule and the time slice, of the program blocks in the cache that are more likely to be purged by intrinsic interference can also be included in the carry over set.

Then, the equation for the miss rate component due to multiprogramming, as shown below, is similar in form to Equation 2 which was derived for non-stationary behavior. The main differences are: (1) the working set size is replaced by the size of the carry over set, $v_i(B)$; (2) $f_{i,u}$, the fraction of references renewed, is replaced by $f_{i,v}$, the fraction of references purged; and (3) the time parameter is adjusted to add in the extrinsic misses once every time slice, t_s.

$$m_i(C,t_i)_{extrinsic} = \frac{v_i(B) f_{i,v}(mpl)}{t_i \tau} \lfloor \frac{(t_i - 1)}{t_s} \rfloor \tag{15}$$

We need to derive an expression for $f_{i,v}(mpl)$, the probability that any reference in the carry over set is purged due to extrinsic interference. To model flushing the cache on every process switch, we can set $f_{i,v}(mpl)$ identically to one. For physical caches, or virtual caches with process identifiers, $f_{i,v}$ can be estimated by applying Binomial statistics to the carry over set of process i and the set of references of all intervening processes. Denoting the number of unique blocks of all intervening processes as $u_{i'}(B)$,

$$u_{i'}(B) = \sum_{j=1, j \neq i}^{j=mpl} u_j(B)$$

and applying binomial statistics we derive the probability, $P_{i'}(d)$, that d blocks fall into any given cache set:

$$P_{i'}(d) = \binom{u_{i'}}{d} \left(\frac{1}{S}\right)^d \left(1 - \frac{1}{S}\right)^{u_{i'} - d}$$

Before proceeding with the derivation, a discussion of the replacement issue in the multiprogramming context is necessary. The intrinsic interference model uses random replacement because the order of use of blocks required to model LRU replacement is not available. However, for multiprogramming, LRU replacement is the natural choice, because the order of execution of processes determines the reference order. Consider the case where process i having just relinquished the processor is followed by $mpl - 1$ intervening processes. Clearly, if LRU replacement is used, the references of process i will be purged before those of the intervening processes. In the ensuing derivation, LRU replacement is first used for simplicity, followed by a discussion on how random replacement can be modeled.

Continuing with the model of process i followed by $mpl - 1$ processes, a block of process i can get purged from a give cache set only if the sum of blocks of processes i (say d) and all intervening processes i' (say e) exceeds the set-size D. Also note that the number of blocks of process i purged in any cache set is the minimum of (1) the number of blocks of process i in that set, d, (2) the set-size D, and (3) the difference between the sum of the number of blocks of process i and i', and the set-size D, or $(e + d - D)$. Cases (1) and (2) are trivial, while case (3) deserves comment. This handles the situation where $d <= D$. As many as $D - d$ blocks out of e of the

intervening processes can co-reside in the set. The remaining, $e-(D-d) = e+d-D$, will collide with the blocks of process i. The number of blocks of process i that get purged in any one set is therefore

$$\sum_{d=0}^{d=D} P_i(d) \sum_{e=0}^{e=u_{i'}(B)} MIN(d, D, e+d-D) P_{i'}(e),$$

which is simplified to yield

$$\sum_{d=0}^{d=D} P_i(d) \sum_{e=D+1}^{e=u_{i'}(B)} MIN(d, e+d-D) P_{i'}(e).^{8} \tag{16}$$

The total number of blocks of process i that get purged is the number purged per set times the number of sets S. The fraction of the blocks that are purged out of the carry over set of process i is then,

$$f_{i,v}(mpl) = \frac{S \sum_{d=0}^{d=D} P_i(d) \sum_{e=D+1}^{e=u_{i'}(B)} MIN(d, e+d-D) P_{i'}(e)}{v_i}, \tag{17}$$

which is substituted into Equation 15 to yield the miss rate component due to extrinsic interference.

Random replacement, modeled in slightly different fashion, assumes that all blocks in a fully occupied cache set are equally likely to be purged, irrespective of which process they belong to. Hence, a block of process i in a collision cache set is displaced with probability d/D by random replacement, as opposed to probability one by LRU. Thus the inner summation in Equation 16 will now include an additional factor, d/D. Contrary to intuition, the extrinsic interference miss rate component with random replacement can actually be smaller than that due to LRU replacement. The explanation is that unlike the LRU scheme random replacement can cause extra collisions amongst blocks of the executing process itself before the blocks of previous processes are completely purged from a set, which shifts misses from the extrinsic category to intrinsic. Therefore, even if the expected number of extrinsic interference misses for the given process decreases, the the total number of misses induced by collisions (intrinsic and extrinsic) is still expected to increase. We also conjecture that shifting misses from the extrinsic to intrinsic category will reduce the bursty nature of misses particularly at process switch boundaries.

This concludes the derivation of the cache model in the multiprogramming environment. The overall miss rate is the sum of the four components calculated in Equations 1, 2, 9, and 15:

$$m_i(C, t_i) = \frac{u_i(B)}{\tau t_i}[1 + f_{i,u}(t_i - 1)] + \frac{c(D)}{\tau}\left[u_i(B) - \sum_{d=0}^{d=D} SP_i(d)\right] + \frac{v_i(B)}{\tau t_i} f_{i,v}(mpl) \left\lfloor \frac{(t_i - 1)}{t_s} \right\rfloor \tag{18}$$

[8] A similar equation for two processes was derived simultaneously and independently by Stone and Thiebaut [26].

4 Applications of the model

Our cache model has been applied to study a number of cache organizations. Cache performance for multiprogramming workloads is also analyzed. Three benchmark traces are used in the uniprogramming cache study: interconnect verify (IV1), which is a DEC program to compare two interconnection net lists in VLSI circuits; a microcode address allocator (AL1); and MILS, an instruction level simulator for the MIPS processor designed at Stanford (TMIL1). The first two traces were obtained using an address tracing scheme called ATUM [1], and TMIL1 was obtained by tracing a VAX-11/780 using the T-bit technique. For the multiprogramming case, MUL10, a trace obtained using ATUM showing ten active processes is used. The processes include FORTRAN and Pascal compilations, numerical benchmarks SPICE, LINPACK, and JACOBI, a string search in a file, an assembler, a linker, an octal dump, and a microcode address allocator.

Figures 9 through 11 show plots of miss rates versus cache-size for a variety of block-sizes and degrees of associativity for the three uniprogramming traces, IV1, AL1, and TMIL1. Corresponding cache performance figures obtained through trace driven simulation for both LRU and random replacement are also shown to assess the accuracy of the model. Analytical model results are shown in solid lines, LRU replacement results in dashed lines, and random (or FIFO) replacement in dotted lines. All cache-sizes are in bytes. Mean and maximum errors in miss-rate estimates over all cache-sizes for random replacement are shown in Table 2. We also present the error in hit rate because for very low miss rates the percentage error in miss rate can be very high, but may not significantly affect performance.

Figures 9(a), 10(a), and 11(a) show the miss rate as a function of cache-size for direct-mapped caches. The miss rates for block-sizes of 4 and 16 bytes are shown. Prediction is quite good for all the benchmarks and block-sizes for a direct-mapped cache. The mean error in miss-rate estimates is about 15%. The hit-rate error is much better at about 1% percent as seen from Table 2.

The miss rates for caches with a set-size of two are shown in Figures 9(b), 10(b), and 11(b). Interestingly, the intrinsic interference component of the miss rate becomes insignificant (miss-rate curve bottoms out) after a cache-size of 32K bytes for two-way set-associative caches, while it is important for direct-mapped caches even as large as 128K bytes. In certain portions of the miss-rate curves, the predictions are lower than simulated miss rates for set-size two. This can be traced to an underestimation of the collision rate, c.

Figures 9(c), 10(c), and 11(c) show the variation of miss rate with cache-size for a block-size of 4 bytes and set-sizes of one and two. For benchmarks AL1 and TMIL1, the miss rate does not drop below one percent for caches larger than 8K words implying that most misses are due to start-up and non-stationary effects. In Figures 9(d), 10(d), and 11(d) the block-size is 16 words and set-sizes are one and two. Note that associativity is not particularly useful for small caches.

Figure 12 summarizes the study of cache performance for multiprogramming workloads. To keep our analysis simple, we exclude the effect of shared system code and data between processes. As before, dotted curves represent simulated miss rates and

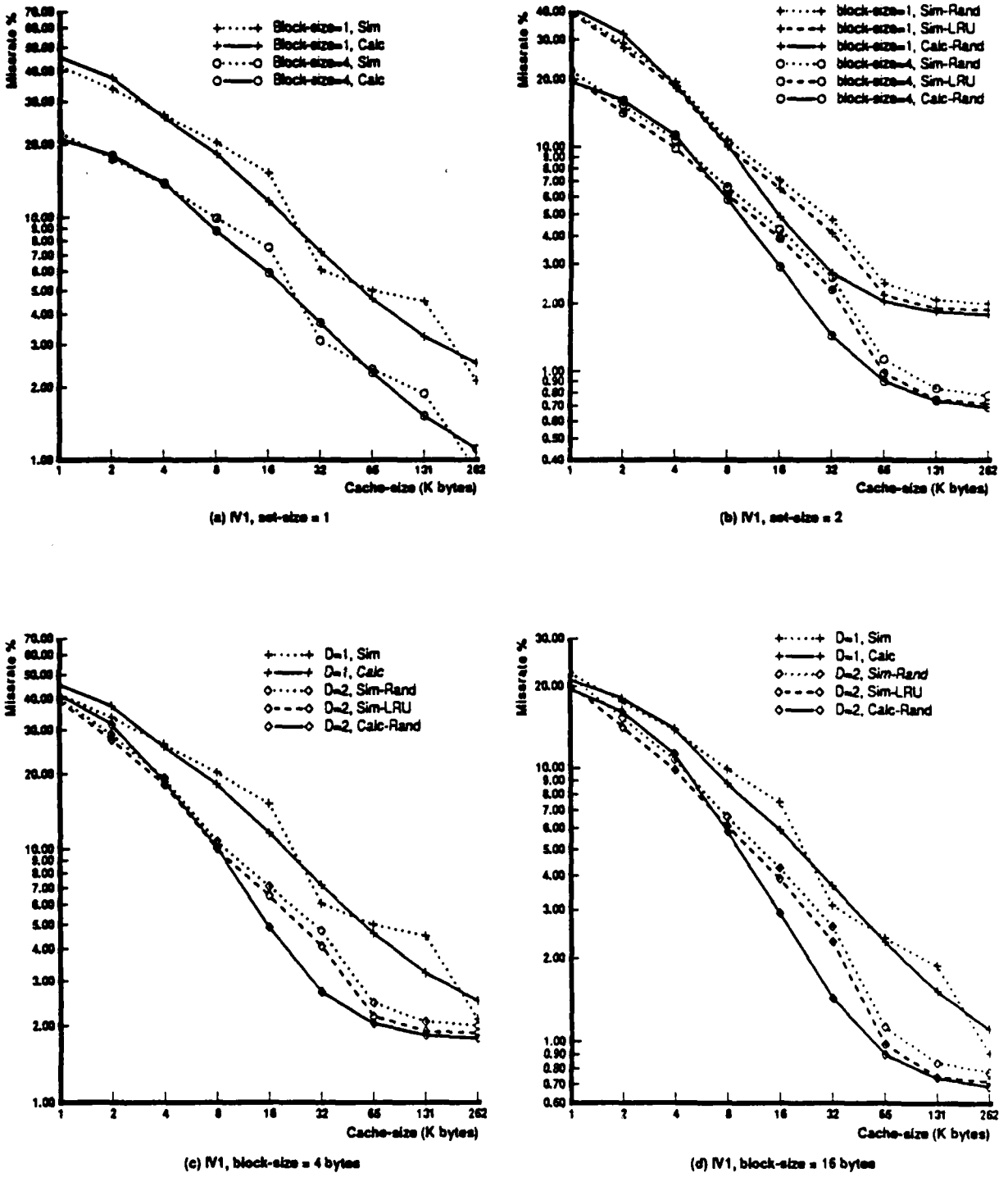


Figure 9: Miss-rate vs. cache-size for the benchmark Interconnect Verify. In (a), (b) set-size is constant, and in (c), (d) block-size is constant. Solid lines represent model calculations, dotted lines represent simulation results for random replacement, and dashed lines correspond to simulation results for LRU replacement.

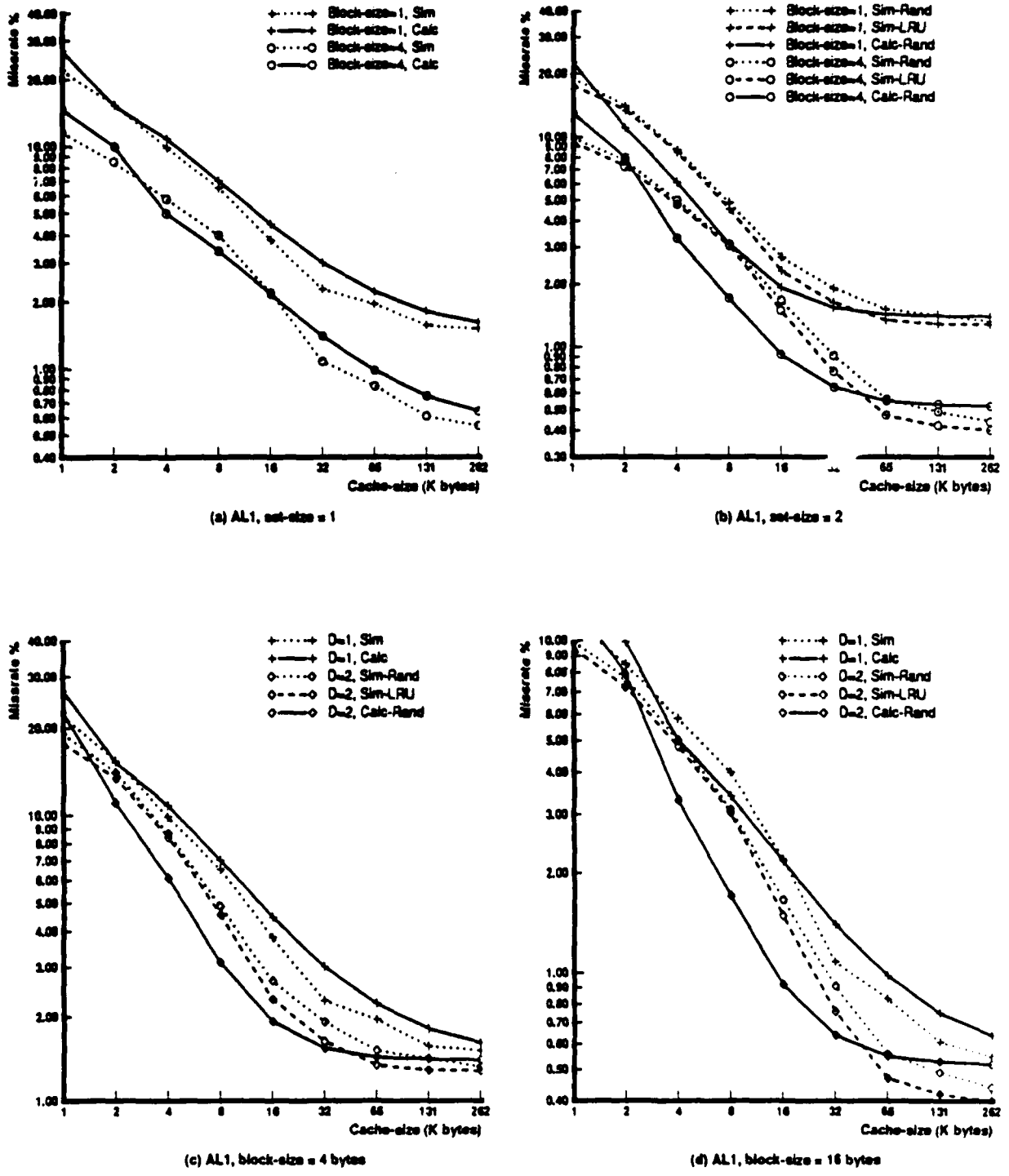


Figure 10: Miss-rate vs. cache-size for the benchmark AL1. In (a), (b) set-size is constant, and in (c), (d) block-size is constant. Solid lines represent model calculations, dotted lines represent simulation results for random replacement, and dashed lines correspond to simulation results for LRU replacement.

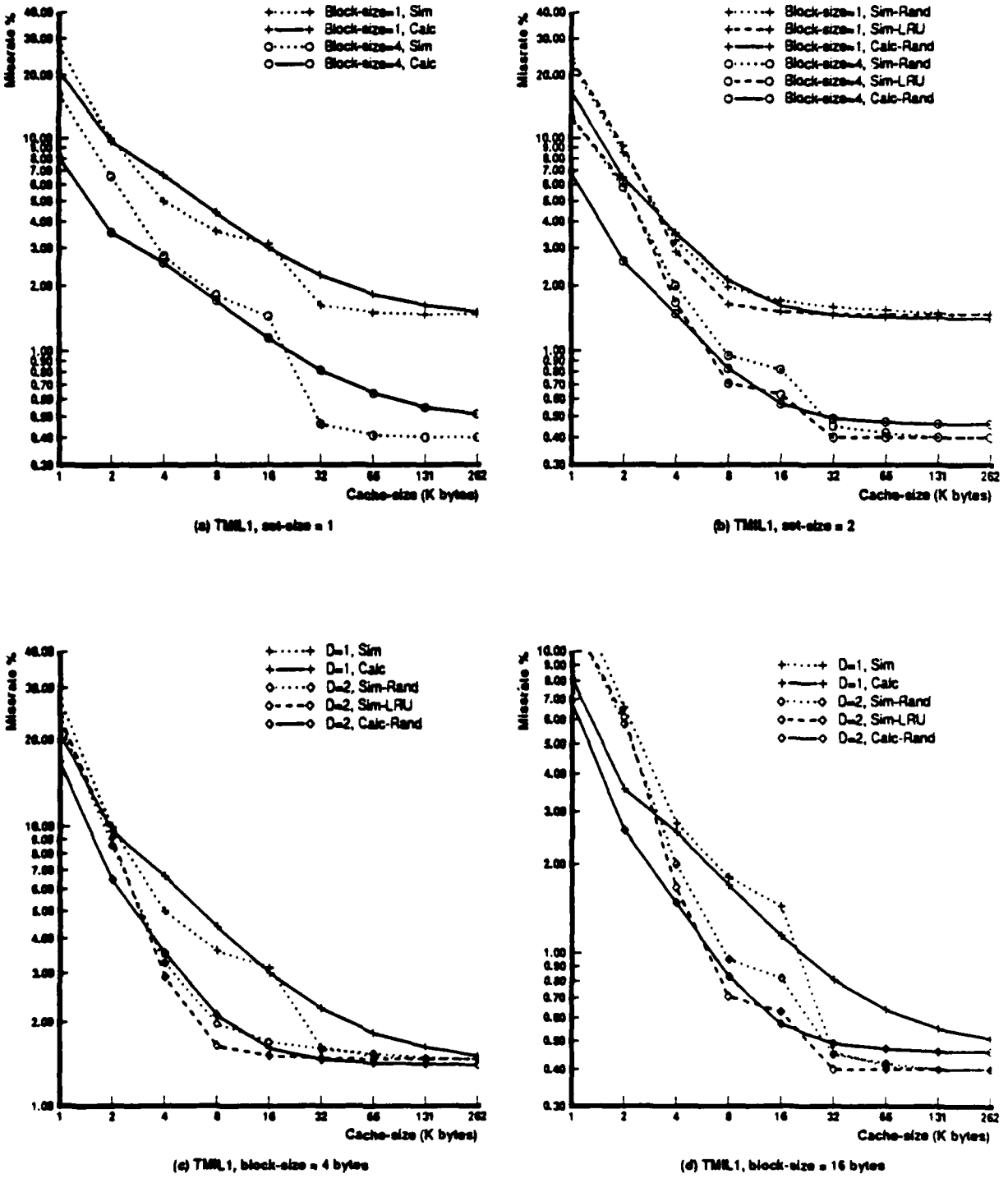


Figure 11: Miss-rate vs. cache-size for the benchmark TMIL1. In (a), (b) set-size is constant, and in (c), (d) block-size is constant. Solid lines represent model calculations, dotted lines represent simulation results for random replacement, and dashed lines correspond to simulation results for LRU replacement.

Setsiz	Blksiz	IV1				AL1				TMIL1			
		MISS		HIT		MISS		HIT		MISS		HIT	
		$\bar{e}$	$\hat{e}$	$\bar{e}$	$\hat{e}$	$\bar{e}$	$\hat{e}$	$\bar{e}$	$\hat{e}$	$\bar{e}$	$\hat{e}$	$\bar{e}$	$\hat{e}$
1	1	14	28	2	7	14	32	1	6	18	38	1	9
1	4	13	22	1	2	19	30	1	4	38	76	2	10
2	1	15	42	1	4	18	36	2	4	11	27	1	8
2	4	19	45	1	3	26	45	1	3	26	55	1	7

Table 2: Percentage error in estimated miss rates and hit rates. $\bar{e}$ and $\hat{e}$ represent mean and maximum errors respectively.

solid curves show estimates. For this experiment, we focus our attention on one process in the trace. The curve with the triangle symbol shows the miss rate for the process assuming that the process has its own separate cache. This is the uniprogramming miss rate for that process. The diamond symbol corresponds to the miss rate of the process assuming that each process is assigned a unique process identifier (PID) which is appended to the tag portion of the address. This method also approximates the miss rate in a physically addressed cache. The bullet symbol depicts the miss rate when the cache is flushed on every process switch. Lacking PIDs, a virtual cache that is flushed on every process switch performs poorly relative to the other schemes for cache-sizes greater than 32K bytes. This is because a significant fraction of the references of a process are reused across process switches. All the schemes perform the same for small caches. Because very large caches can simultaneously hold the working sets of a number of processes, the miss rates of large caches for the PID scheme and for uniprogramming are very similar.

In general, the model predictions are similar to those of trace driven simulation. However, the model does not capture the sharp and often abrupt changes in miss rate that many programs display. For example, in Figure 9(a), IV1 shows a sharp drop in miss rate in going from a cache-size of 16K to 32K bytes, while the predicted behavior shows a smooth trend. The estimated results are most accurate if the trace parameters are measured from a cache in a close vicinity of the cache types of interest. This provides a useful strategy to obtain fine tuned results. For example, parameters could be measured keeping the block-size fixed throughout the analysis.

In general, parameter extraction is reasonably straightforward, but may require analyzing the entire trace if maximum accuracy is desired; parameters can otherwise be extracted from sample segments of the trace. While u and U – the average number of unique references in a time granule and in the entire trace respectively – are easily measured, some of the other parameters deserves more comment. The collision constant, c, is obtained by simulating a typical cache. If the cache is small enough, the entire trace need not be necessary because once steady state is reached only a few more references need be simulated to give a good indication of c. The spatial locality parameter, f_{l1}, is simply the fraction of singular addresses in the trace; f_{l2} can be derived using Equation 10 and the measured average run-length (ratio of the number of runs to the number of references in the trace). Depending on the desired accuracy in multiprogramming results, either an average time slice parameter or a distribution

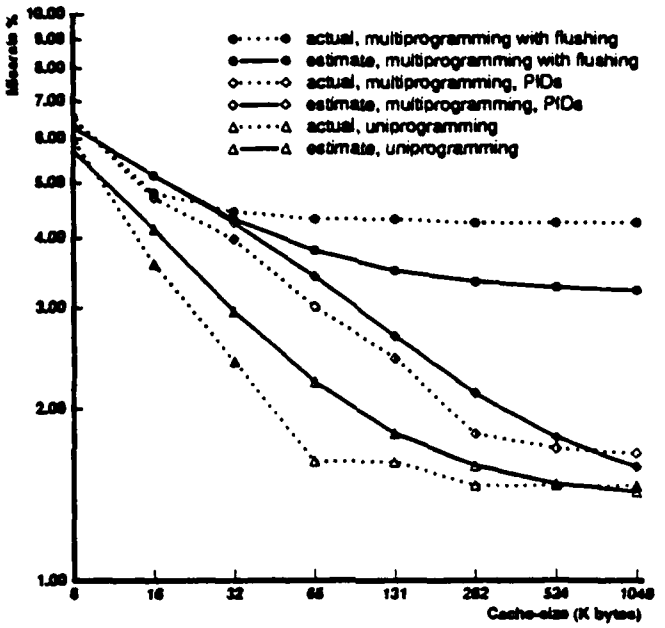


Figure 12: Miss-rate vs. cache-size (K bytes) for multiprogramming. Block-size is 16 bytes and cache is direct-mapped.

(measured or assumed) can be used.

We concentrated on the effects of various cache parameters on the miss rate. Analyzing the sensitivity of the miss rate on various program and workload parameters, another interesting area, is the subject of ongoing research.

5 Conclusions

An analytical model for caches driven by a few parameters measured from program traces has been presented. A judicious combination of measurement and analytical techniques reduce computation time without significantly sacrificing accuracy. Cache performance due to start-up effects, gradual locality changes in program execution, contention for a cache block, and multiprogramming can be quickly estimated for most cache parameters of interest, including cache-size, block-size, degree of associativity, trace-size, and multiprogramming level. Furthermore, explicitly displaying the sensitivity of the miss rate on various program and workload parameters helps identify areas in which further research in improving cache performance would be fruitful.

6 Acknowledgements

This work has been supported by Defence Advanced Research Projects Agency under contract # MDA903-83-C-0335.

We are grateful to Ed Lazowska for some stimulating discussions at the start of this research, and to Mark Hill and Susan Eggers for reading a draft of the paper and providing useful feedback.

Appendices

A Sensitivity of the miss-rate on time granule τ

The choice of τ in the cache model has been rather ad hoc; we now examine the sensitivity of the miss-rate for uniprogramming to this choice. We will concentrate only on the intrinsic interference component because the sum of the start-up and non-stationary components of the of the miss-rate for the entire trace is simply the ratio of the total number of unique references to the total length of the trace and hence does not depend on the choice of the time granule τ. The choice of τ directly affects the parameter u, which is the average number of unique references in a time granule.

The following equation gives the intrinsic miss-rate component for direct mapped caches:

$$m(C,t)_{intrinsic} = \frac{c[u(B) - S\ P(1)]}{\tau}$$

Replacing the collision rate and $P(1)$ by their respective formulae and gathering the factors that are independent of u into the constant K shows the complete dependence of the intrinsic miss-rate component on u:

$$m(C,t)_{intrinsic} = K\ \frac{u(B)}{u(1)}\ \frac{1 - \left(1 - \frac{1}{S}\right)^{u(B)}}{1 - \left(1 - \frac{1}{S_0}\right)^{u(1)}}$$

If τ is chosen to be at least as large as the start-up portion of the trace, then variations in τ will not affect the miss-rate significantly because u changes very gradually after the start-up region, and the ratio of $u(B)$ and $u(1)$ will be relatively stable.

Since the discussion hinges on the definition of a *start-up* period, we will digress a little to analyze some of the common notions of start-up time. Recall that for the purpose of our model the start-up period in a trace is the time (or number of references) required to bring the initial working set of the program into the cache for the first time. This definition is solely a function of the program and independent of the cache organization. Because the initial working set of a program is hard to quantize precisely, estimating the start-up period in a trace is non-trivial.

The *cold-start* period is a related term and was defined by Easton and Fagin [8] in the context of a fully associative cache to be the number of memory references from an initially empty cache until the number of misses equals the cache-size in blocks. Although this definition allows precise measurement of the cold-start period, it is inappropriate in our case because (1) it depends on cache-size, (2) in small caches start-up misses can occur even after the cache has been heavily filled, and (3) in large caches, non-stationary misses can occur long before the cache is filled. Furthermore, the notion of a cache filling up is relevant only to fully associative caches.

A pertinent definition for the start-up period that is not a function of cache-size, but solely dependent on the address trace in question is based on the working set model [7] of program behavior.[9] The number of unique blocks used by a process in a time granule increases rapidly as the time granule is increased from zero to some value and increases only gradually thereafter causing working set curves to have a bilinear nature (please see [25]). We will define the *start-up portion* to be the region of the working set curve before its knee point. The latter part will then represent the non-stationary region. Figure 13 shows $u(1)$ as a function of time granule size τ for the benchmark Interconnect Verify. The knee occurs between ten and fifteen thousand references. The dotted line, which is the derivative of the working set curve, is the number of additional blocks accessed for a given increase in granule size. After about 10,000 references the increase in u is small and steady.

There is one caveat, however, in this discussion. For phased programs the working set curve will show slightly different behavior, although the nature of the curve in any one phase is still expected to be the same provided the phases are long enough for the working set model to apply. As before, the start-up period is measured from the working set of the initial phase, and the rest of the first-time misses are swept into the non-stationary category. Traces showing multiple short phases (e.g., for inter-active workloads) may theoretically exhibit very long start-up times; but for such behavior a large portion of the misses will be start-up induced anyhow.

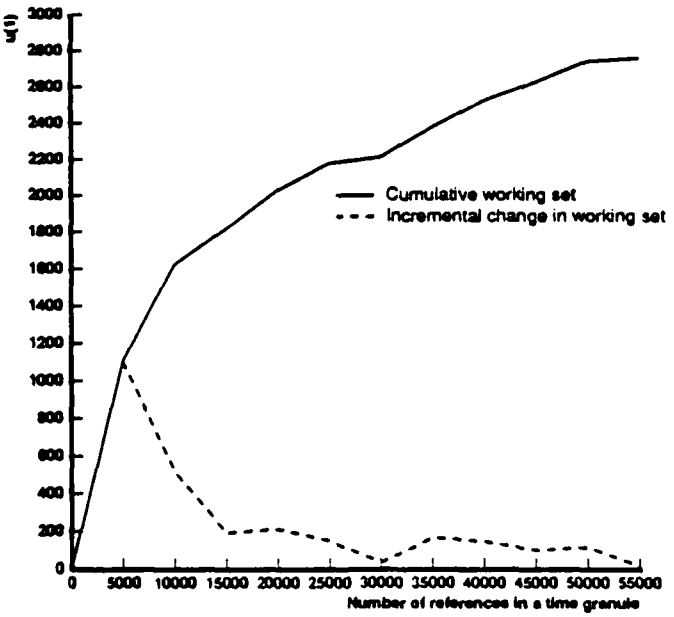


Figure 13: Number of unique references per time granule, $u(1)$, versus granule size τ.

Returning to our earlier discussion on the stability of the collision rate, once τ is

[9]Cold-start period for large non fully-associative caches, which is also an interesting issue, is not further addressed this paper.

greater than 10,000 references, increasing it further will cause little change in $u(1)$ and still less in the ratio of $u(B)$ to $u(1)$. Thus, we have showed that choosing τ greater than the start-up period will cause the intrinsic miss rate to be insensitive to changes in τ. For smaller values of τ, u varies enormously and potential for large errors exist. However, since changes proportional to $u(B)$ and $u(1)$ are expected in the numerator and denominator of the intrinsic miss-rate equation, the differences should cancel out to first order and the miss-rate should be reasonably stable.

B Characterization of the collision rate c

The intrinsic interference model uses the collision rate c – the ratio of the total number of collisions to the number of colliding blocks – to quantize the dynamic interference component among program blocks. The product of the static number of colliding blocks and the average number of times a block collides, c, gives the average number of misses in the cache due to intrinsic interference in a time granule. The thesis is that c is reasonably stable for caches of different numbers of sets (rows) and block-sizes (line-sizes). The collision rate, however, varies with set-size (associativity or number of columns) and is not stable when the cache becomes much smaller than the size of the program working set. This section has the following parts to address these issues:

1. Provides an intuitive basis and some measured data for the assumption that c is stable for different cache sizes.
2. Extends c to all cache organizations.

B.1 On the stability of the collision rate

We present the following argument as as intuitive substantiation of the claim that c is constant for most cache sizes and organizations. This is also verified by our measurement data, a sample of which is presented in Table 3 for Interconnect Verify. A direct mapped cache with block-size of 4 bytes is assumed in the ensuing discussion unless otherwise stated.

Recalling, a *collision set* is a set with multiple blocks mapped to it, and a *colliding block* is a block that maps into a colliding set. As an illustration, Figure 14 shows a direct-mapped cache with 8 sets, $S0$ through $S7$. A block is denoted by a shaded rectangle. The number of collision sets are two ($S3$ and $S6$) and the number of colliding blocks are five ($B1$, $B2$, $B3$, $B5$, and $B6$). The average number of collisions per block in a collision set will be an increasing function of the number of blocks present in that set. Clearly, the more the number of blocks in a collision set the greater the probability that a block will be purged by intervening references, and hence greater the collision rate. Therefore, c will show similar behavior to the parameter c' that we define to be the number of colliding blocks in an average collision set. Note that c' is not used to estimate c. In the cache shown in Figure 14, $c' = 2.5$.

Equation 3 gave the probability that a set has d blocks mapped to it. We repeat the equation here making the approximation that the binomial distribution tends to the Poisson for large u_i and small $(1/S)$:

$$P_i(d) = \frac{e^{-\frac{u_i}{S}} \left(\frac{u_i}{S}\right)^d}{d!} \tag{19}$$

As derived earlier, the number of colliding blocks is $u_i - SP(1)$ and the number of colliding sets is $S(1 - P(0) - P(1))$, yielding,

$$c' = \frac{u_i\left(1 - e^{-\frac{u_i}{S}}\right)}{S\left(1 - e^{-\frac{u_i}{S}} - \frac{u_i}{S}e^{-\frac{u_i}{S}}\right)}.$$

The following plot (Figure 15) gives the variation of the above function. We also plot the collision rate, c, to check the correspondence. Clearly, c' (and hence a corresponding c) is stable for cache sizes as low as 512 words because, above this value, both the number of collision sets and colliding blocks decrease in the same proportion with cache-size. Below 512 words, c' increases rapidly (c shows a similar anomalous upward trend) because the denominator starts to decrease in proportion to S and the numerator stays constant at u_i. Thus, c can be expected to remain stable if the cache-size is greater that the working-set estimate, u_i. In addition, changing block-size does not effect c for large caches as can be verified from Table 3. The rationale is that the dynamic behavior of program blocks is statistically similar to that of their component words. For instance, if two words collide with each other in the cache, then the two blocks that contain the words will also collide at the same rate. The collision rate does depend on the set-size and on cache-size for small caches. This will be the subject of our discussion in the next section.

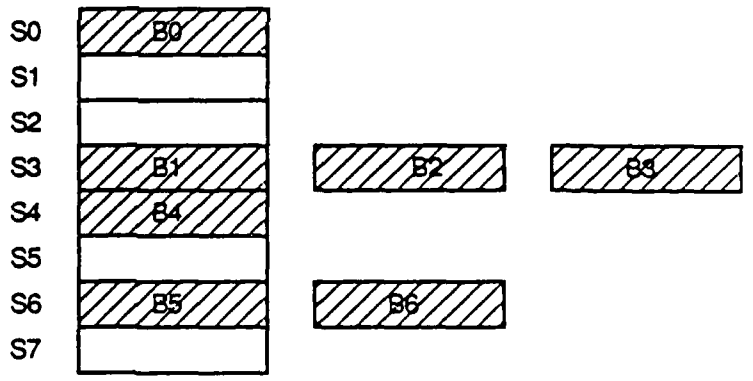


Figure 14: Collision in a cache.

B.2 Estimating variations in the collision rate

We first derive c for non-unit set-sizes. Let $c(D)$ denote the value of c for a cache with set-size D. In a direct-mapped cache, a reference to any but the most recently referenced address in the set will cause a miss. For a larger set-size, however,

S	D	B	Coll-blks	Misses-LRU	Misses-FIFO	c-LRU	c-FIF	c(est)
Vary Number of Sets								
32	1	1	1624	6859	6859	4.2	4.2	4.0
64	1	1	1624	5658	5658	3.5	3.5	3.6
128	1	1	1624	5049	5049	3.1	3.1	3.1
256	1	1	1622	3908	3908	2.4	2.4	2.7
512	1	1	1557	3106	3106	2.0	2.0	2.3
1024	1	1	1282	2391	2391	1.9	1.9	1.9
2048	1	1	914	1798	1798	2.0	2.0	1.9
4096	1	1	609	1305	1305	2.1	2.1	1.9
8192	1	1	254	413	413	1.6	1.6	1.9
16384	1	1	161	308	308	1.9	1.9	1.9
Vary Set-size								
1024	1	1	1282	2391	2391	1.9	1.9	1.9
512	2	1	1355	1593	1680	1.2	1.2	1.3
256	4	1	1418	1404	1511	1.0	1.1	1.0
16384	1	1	161	308	308	1.9	1.9	1.9
8192	2	1	30	32	42	1.1	1.4	0.9
4096	4	1	8	10	19	1.2	2.3	0.5
Vary block-size								
1024	1	1	1282	2391	2391	1.9	1.9	1.9
512	1	2	908	1784	1784	2.0	2.0	1.9
256	1	4	607	1268	1268	2.1	2.1	2.3
128	1	8	410	1017	1017	2.5	2.5	2.4
64	1	16	283	1037	1037	3.7	3.7	2.9
16384	1	1	161	308	308	1.9	1.9	1.9
8192	1	2	126	230	230	1.8	1.8	1.9
4096	1	4	93	163	163	1.7	1.7	1.9
2048	1	8	74	122	122	1.6	1.6	1.9
1024	1	16	65	102	102	1.6	1.6	1.9

Table 3: Measured variation in the collision rate for IV. Coll-blks represents the number of colliding blks, Misses-LRU and Misses-FIFO the number of collision induced misses for LRU and random replacement respectively, c-LRU and c-FIFO, the measured values of c for LRU and FIFO replacement, followed by the estimated value of c.

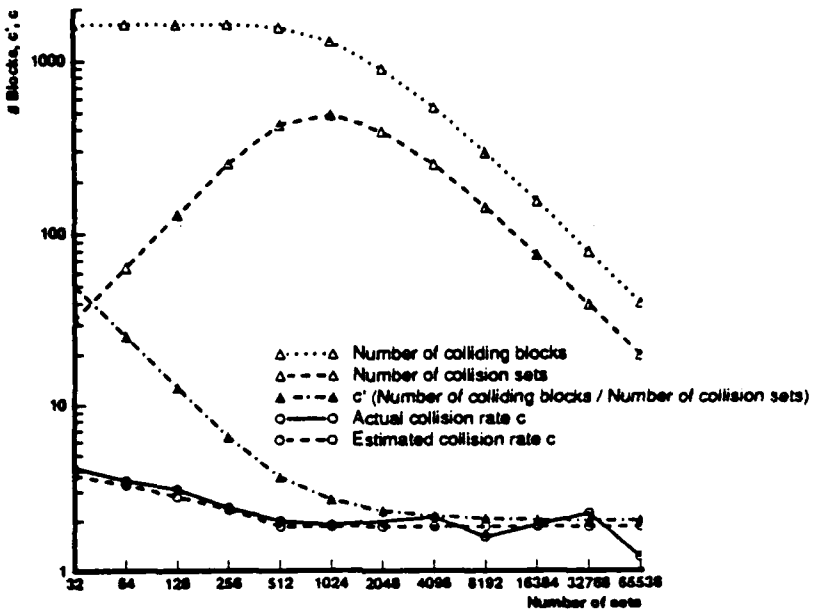


Figure 15: Estimating the collision rate c.

reference to as many as $D-1$ blocks beside the most recently referenced one may not cause a miss. Assuming random behavior, the probability of a hit to any one of the d blocks mapped into the set is $(D-1)/(d-1)$. Note that the most recently referenced block is excluded because its effect has already been included in c. The corresponding probability of a miss is one minus the above quantity. Therefore, $c(D,d)$, the collision rate for a set of size D with d overlapping blocks is $c(1)$ weighted by this fraction, where, as before, $c(1)$ is the collision rate for a direct-mapped cache with the same number of sets.

$$c(D,d) = c(1)\left(1 - \frac{D-1}{d-1}\right)$$

We can then obtain an average of $c(D,d)$ over all d's as:

$$c(D) = \frac{\sum_{d=D+1}^{d=\infty} c(D,d)dP(d)}{\sum_{d=D+1}^{d=\infty} dP(d)}.$$

Table 3 shows the variation in c for the benchmark IV1. Estimated values of c and also measured values using both LRU and random replacement are provided. The number of colliding blocks and the actual number of collision induced misses for LRU and FIFO replacement schemes are also shown. S, D, and B, represent the number of sets, set-size and block-size respectively. c seems relatively stable for direct mapped caches when the cache size is greater than half $u_i(1)$. However, predictions of c for non-unit set-sizes are often optimistic. Note that even large errors in estimating c will not affect the miss-rate if the number of colliding blocks is very small. For example, a

64K byte cache organized with an associativity of 4, FIFO replacement, and block-size of 4 bytes, yields 8 colliding blocks, and only 19 collision induced misses for IV1. The 78% error in the collision rate estimate clearly causes little error in the 2% miss-rate.

We now provide a method of estimating the collision rate when the number of sets is much smaller than the number of program blocks. This region of the cache organization spectrum is only of marginal interest and hence this discussion is provided mainly for the sake of completeness. We had shown earlier that c is bounded as $0 \leq c \leq \tau/u$, the maximum being attained when the cache has only one set. Furthermore, Figure 15 shows that c' is inversely proportional to S for very small caches. This leads to a rough approximation of c.

We denote the stable value of c by c_0 measured for the representative cache with $S = S_0$. Let S_{u_i} to be the power of 2 less than and closest to u_i. When the number of sets falls below S_{u_i}, all the sets will be occupied with a high probability. Then, empirical results show that c can be increased in inverse proportion to the log of the number of sets up to its maximum value for $S = 1$ as:

$$c = c_0 + \left[\left(\frac{\tau}{u_i}\right) - c_0\right]\left(1 - \frac{log(S)}{log(S_{u_i})}\right) \tag{20}$$

Figure 15 shows the measured and calculated values of c. For number of sets less than 512 the log approximation appears to be quite good.

C Inter-run intervals and spatial locality

We showed earlier that the distribution of run-lengths alone is not sufficient to characterize the spatial locality in programs especially for block-sizes in excess of 8 words. We address this issue in this discussion by calculating the number of blocks needed to contain the reference stream in a slightly modified fashion to account for blocks that contain portions of more than a single run. The initial derivation assumes that the entire inter-run interval distribution is available, but later we will present a simplified formulation that requires just the average inter-run interval and gives comparable results.

As before, cover-size for a run is the average number of blocks that have to be fetched to bring the entire run into the cache. Due to random alignment of the run, $B-1$ words in the cover are unused by the run on average; allocating all these extra words to the run inflates working set estimates as shown in Figure 8. The number of blocks in a cover actually allocated to contain the run[10] is

$$
\begin{aligned}
&= \textit{Number of blocks needed to cover that run} \\
&\quad - \textit{number of blocks that include neighboring runs} \\
&= \textit{Cover size for the run} \\
&\quad - \frac{\textit{Number of words that include neighboring runs}}{\textit{Blocksize}}
\end{aligned}
$$

If the run-length is l and the block-size is B, the number of words in the cover (or set of blocks that contain the run) that do not contain any valid part of the run in question is $B-1$. Let $exc = B-1$ denote the excess number of words and let $N(exc)$ denote the number of words out of exc that are utilized for neighboring runs. This means that the number of blocks actually used up by the given run is

$$\left[1 + \frac{l-1}{B} - \frac{N(exc)}{B}\right]$$

and the total number of blocks as calculated before (see Equation 12) is

$$u(B) = \frac{u(1)}{l_{av}} \sum_{l=1}^{l=\infty} R(l) \left[1 + \frac{l-1}{B} - \frac{N(exc)}{B}\right] \tag{21}$$

The total number of unique blocks in the entire trace, $U(B)$, is also calculated as above. We now need to derive the function $N(exc)$. Let $I(intvl)$ be the distribution of inter-run interval lengths. Then, the expected number of words out of those left over in the cover (exc) that are used in covering *the first adjacent run* is given by,

$$\sum_{intvl=1}^{intvl=n} I(intvl) \sum_{l=1}^{l=\infty} R(l) MIN(l, exc - intvl)$$

where,

[10] Including both both forward and backward neighbors.

$I(intvl)$ = probability that an interval is of length $intvl$, and

$R(l)$ = probability that a run is of length l.

Note that the maximum number of addresses of any run that can be covered is $exc - intvl$. $N(exc)$ can then be calculated in a recursive fashion, as follows:

$$N(exc) = \sum_{intvl=1}^{intvl=exc} I(intvl) \sum_{l=1}^{l=\infty} R(l)\,[MIN(l, exc - intvl) + N(exc - intvl - l)] \quad (22)$$

with $N(exc \leq 1) = 0$.

It is also reasonable to sacrifice some accuracy and lump the intervals into two values, I_{av} and ∞, where all interval sizes up to the maximum block-size of interest are averaged to give I_{av}, and the remaining are categorized as ∞. Then,

$$N(exc) = f_{I_{av}} \sum_{l=1}^{l=\infty} R(l)\,[MIN(l, exc - I_{av}) + N(exc - I_{av} - l)], \quad (23)$$

and $N(exc \leq 1) = 0$

In the above equation, $f_{I_{av}}$ is the fraction of intervals that are averaged to give I_{av}. In the implementation, it is worthwhile to pre-calculate $R(l)$ and $N(exc)$ and store the values in arrays and use dynamic programming to eliminate recursion.

The total number of unique program blocks in a time granule, $u(B)$, and those in the entire trace, $U(B)$, can then be calculated as shown in Equation 21 using the above formulation of $N(exc)$. Figure 16 shows the variation in $u(B)$ as a function of block-size to compare actual and predicted values. The predicted values are obtained using a measured average inter-run interval. These were not appreciably different from those calculated using the complete distribution of inter-run intervals. The variation in $u(B)$ using just run-length distributions is also shown. The difference between the dotted and the dashed curves can be attributed to the capture of multiple runs separated by inter-run intervals. The mean and maximum errors for the estimate using inter-run intervals are 19% and 64%, while those for the estimate excluding inter-run intervals are 33% and 115%.

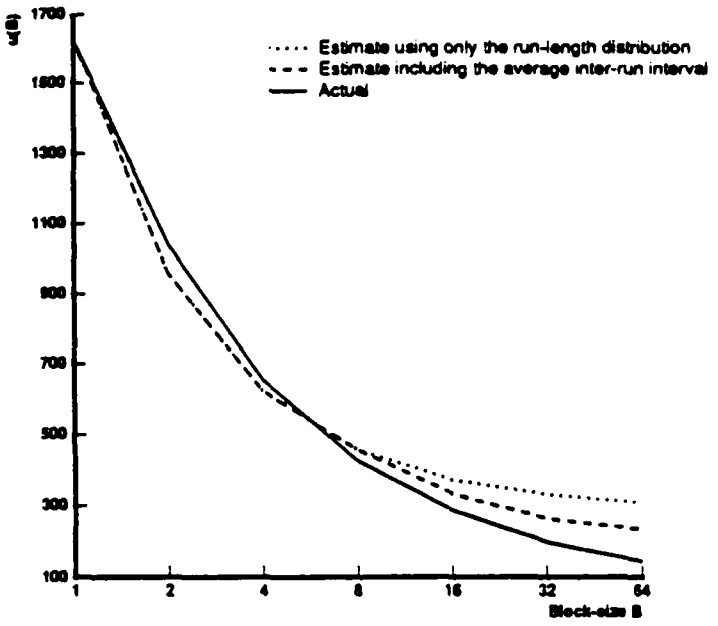


Figure 16: The average number of unique blocks in a time granule, $u(B)$, versus the block-size B. The estimate using inter-run intervals is more accurate that with just the run-length distribution.

References

[1] Anant Agarwal, Richard L. Sites, and Mark Horowitz. Atum: a new technique for capturing address traces using microcode. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 119–127, June 1986.

[2] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *JACM*, 18(1):80–93, January 1971.

[3] Donald Alpert. *Performance Tradeoffs for Microprocessor Cache Memories.* Computer Systems Laboratory 83-239, Stanford University, December 1983.

[4] David R. Cheriton, Gert A. Slavenberg, and Patric D. Boyle. Software-controlled caches in the nebula multiprocessor. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 367–374, June 1986.

[5] D. W. Clark and J. S. Emer. Performance of the vax-11/780 translational buffer: simulation and measurement. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[6] Douglas W. Clark. Cache performance in the vax-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, February 1983.

[7] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[8] M. C. Easton and R. Fagin. Cold-start vs. warm-start miss ratios. *Communications of the ACM*, 21(10):866–872, October 1978.

[9] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, June 1983.

[10] Ilkka J. Haikala. Cache hit ratios with geometric task switch intervals. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 364–371, June 1984.

[11] Mark Hill and Alan Jay Smith. Experimental evaluation of on-chip microprocessor cache memories. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 158–166, June 1984.

[12] P. G. Hoel, S. C. Port, and C. J. Stone. *Introduction to Stochastic Processes.* Houghton Mifflin Company, Boston, 1972.

[13] Chow C. K. Determining the optimum capacity of a cache memory. *IBM Technical Disclosure Bulletin*, 17(10):3163–3166, March 1975.

[14] B. Kumar. A model of spatial locality and its application to cache design. 1979. Unpublished Report, Computer Systems Laboratory, Stanford University.

[15] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance.* Prentice Hall, 1984.

[16] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[17] G. S. Rao. Performance analysis of cache memories. *JACM*, 25(3):378–395, July 1978.

[18] B. R. Rau. *Sequential prefetch strategies for instructions and data.* Technical Report 131, Digital Systems Laboratory, Stanford University, January 1977.

[19] Jerome H. Saltzer. A simple linear model of demand paging performance. *Communications of the ACM*, 17(4):181–186, April 1974.

[20] Alan Jay Smith. Cache evaluation and the impact of workload choice. In *Proceedings of the 12th Annual Symposium on Computer Architecture*, pages 64–73, June 1985.

[21] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[22] Alan Jay Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, SE-4(2):121–130, March 1978.

[23] Alan Jay Smith. *Line (Block) Size Choice for CPU Cache Memories.* Computer Science Division 85-239, University of California, Berkeley, June 1985.

[24] James E. Smith and James R. Goodman. Instruction cache replacement policies and organizations. *Proceedings of the 10th Annual Symposium on Computer Architecture*, C-34(3):234–281, March 1985.

[25] J. R. Spirn. *Program Behavior: Models and Measurements. Operating and Programming Systems Series*, Elsevier, New York, 1977.

[26] Harold S. Stone and Dominique Thiebaut. Footprints in the cache. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 4–8, May 1986.

[27] William D. Strecker. Transient behavior of cache memories. *ACM Transactions on Computer Systems*, 1(4):281–293, November 1983.

[28] Kishore. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* Prentice Hall, 1982.

END

7-87

DTIC